2019

Anticipation in Dynamic Environments: Deciding What to Monitor

Zohreh A. Dannenhauer
Wright State University

ANTICIPATION IN DYNAMIC ENVIRONMENTS: DECIDING WHAT TO MONITOR

A Dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

by

ZOHREH A. DANNENHAUER
B.S., Tarbiat Moallem University of Tehran, 2009
M.S., Wright State University, 2018

2019
Wright State University

Wright State University
GRADUATE SCHOOL

April 29, 2019

I HEREBY RECOMMEND THAT THE DISSERTATION PREPARED UNDER MY SUPERVISION BY ZOHREH A. DANNENHAUER ENTITLED ANTICIPATION IN DYNAMIC ENVIRONMENTS: DECIDING WHAT TO MONITOR BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Doctor of Philosophy.

Michael T. Cox, Ph.D.
Dissertation Co-Director

Michael L. Raymer, Ph.D.
Dissertation Co-Director

Michael L. Raymer, Ph.D.
Director, Computer Science and Engineering PhD Program

Barry Milligan, Ph.D.
Interim Dean of the Graduate School

Committee on
Final Examination

Michael T. Cox, Ph.D

Michael L. Raymer, Ph.D

Pascal Hitzler, Ph.D

Michelle A. Cheatham, Ph.D

Hector Munoz-Avila, Ph.D

Matthew Molineaux, Ph.D

ABSTRACT

Dannenhauer, Zohreh A. Ph.D., Computer Science and Engineering Department, Wright State University, 2019. Anticipation in Dynamic Environments: Deciding What to Monitor.

In dynamic environments, external changes may occur that may affect planning decisions and goal choices. We claim that an intelligent agent should actively watch for what can go wrong and anticipate changes in the environment that allows the changing of its plan or changing of a given goal. In this thesis, we focus on the relationship between perception, act, interpretation and planning. We claim that these components are not independent and need to interact with each other to help the agent succeed in achieving its goals and plans. If newly encountered world information affects the plan, the agent adapts to it through the refinement of plans under construction. If the justification for goal selection changes, then the agent should transform or abandon the goal. Our approach is to make perception sensitive to relevant changes in the environment that can affect plans and goals. We will present results with a cognitive architecture in different domains such as blocksworld, logistics, minecraft and a Baxter humanoid robot to show the effectiveness of the proposed approach.

Contents

List of Figures

List of Tables

Acknowledgment

I would like to take this opportunity to thank my adviser, Mike Cox, who encouraged me along the path and provided guidance over the past few years. He always inspired me with his hard work and his way of thinking about problems. Mike was the one who first introduced me to cognitive architectures and goal reasoning. He asked me to work on a Baxter robot and the MIDCA cognitive architecture when I started working with him. From that point, my research contributions fell within the context of goal reasoning and MIDCA, and more broadly, autonomous agents. Finally, I would like to thank him for spending a great amount of time reviewing the draft of this thesis and my other publications. I also would like to thank Matt Molineaux for his helpful practical suggestions and constant availability for motivating discussions. I also would like to thank the rest of my committee: Michelle Cheatham, Pascal Hitzler, Hector Munoz-Avila, and Michael Raymer for their valuable feedback on my proposal.

I additionally, thank Danielle Brown for her help in reviewing the draft of this thesis and providing helpful feedback. Also, I am grateful for having constructive discussions in our group meetings with Sravya Kondrakunta and Sampath Gogineni. I also appreciate Vahid Eyorokon and Mak Roberts for their constructive comments and suggestions on drafts of my previous publications. I would like to thank my undergraduate AI professor Dr. Mohsen Pedram who first introduced me to AI and encouraged me to apply for graduate school.

ONR supports this research under grants N00014-15-1-2080, N00014-15-C-0077 and N00014-18-1-2009 and AFOSR provides support through grant FA2386-17-1-4063.

Finally, I would like to thank my family for their support. My husband, Dustin Dannenhauer contributed with his support and love. He always knows how to put me back on my original goal and supports me in all the ups and downs of my studies. I also thank my parents in Iran for their endless love and support and for enduring all these years in graduate school that I could not see them. They helped me to come to the US, and if it wasn't for them I could not have started my Ph.D. program.

This thesis is dedicated to

my parents and my husband

Introduction

The ability to act and respond to exogenous events or actions in dynamic environments is crucial for robust autonomy. Real world autonomous agents will need to manage their behavior across many complex situations and to solve severe problems that arise while pursuing their goals. In dynamic environments, unexpected external changes may occur that prevent an agent from reaching its goal(s). These external changes may cause discrepancies between the agent's expectations and observations. We claim that an intelligent agent should actively anticipate mistakes before they occur in order to successfully complete its missions.

We focus on general autonomous agents associated with *cognitive architectures*. All cognitive architectures share some common features. First, cognitive architectures model a combination of cognitive processes such as action, perception and memory. Second, these cognitive architectures provide domain-independent and fixed infrastructures. With the addition of domain-specific knowledge, cognitive architectures can solve a variety of tasks. Last, they are meant to show human-like intelligence including showing robust behavior when they face unexpected situations [54].

A cognitive agent is an intelligent system that uses knowledge structures to reason about the goals to which it is committed in an environment within which it can perceive and act [44]. To achieve its goals, such an agent requires several reasoning capabilities. The first such capability is planning, which constructs a sequence of actions that transform the initial state to the goal state. However, in a dynamic environment, the agent must monitor

its plans since external events can occur and refine its plan when the relevant features change. The second such capability is interpretation which analyzes the percepts in terms of the agent's existing knowledge to ensure the agent is making progress toward its goals. Cognitive agents may change their goals when an anomalous situation arises. Often times, the anomaly prevents the agent from reaching its original goal and to achieve the original goal, the agent must remove the cause of the anomaly. To remove the cause of the anomaly, the agent may pursue a new goal, which upon completion, will allow the agent to return to pursuing its original goal. In a partially observable environment, an agent also should be able to explain anomalous state changes by inferring root causes that were not observed [59].

Consider the following example in the game of Minecraft. Steve (our Minecraft agent) has a goal to have some wood. The plan to achieve his goal has the following steps: move to the location of a tree; change the tool to pickaxe; harvest the tree; and gather wood. While working toward his goal, Steve's health decreases, and Steve finds this anomalous because he is expecting good health (a health-value $\geq$ 30). Instead he observes a lower health value. Low health is the consequence of some event that has occurred. The agent does not have full knowledge of his surroundings, but he hypothesizes that he was shot by an archer skeleton or activated an arrow trap. He may stop his current goal to address this more immediate problem.

The kinds of problems cognitive agents face have increasingly become those in which the agent's goals must be flexible given the dynamic nature of the environments within which they operate. This is a different assumption than previous approaches that assume goals are static and received from an external user. The agent itself is expected to recognize situations in which new goals are to be formulated or current goals changed and abandoned. This is the basis for an area of research called *goal reasoning* [1, 2, 40, 68].

Goal reasoning agents treat goals as dynamic structures that can be managed in a variety of ways such as formulating, selecting, suspending, delegating [22], and rejecting

[25]. In some cognitive agents, there are different rules that relate the agent's belief to goals [16]. These rules control the agent's behavior and its ability to formulate and select the goals based on the description of the world. The state not only triggers the goal in the agent, it also provokes the agent to abandon or change its goal. When a particular situation disappears or a new one arises the agent may need to stop what it has been doing or start working toward a new goal.

This thesis focuses on the relationship between planning, acting, interpretation, and perception in a cognitive architecture. We claim these cognitive processes should interact intelligently to make the agent robust and effective in dynamic worlds. During planning, perception should notify the planner about the relevant changes in the world. During perception and interpretation, the planner should guide perception and interpretation mechanisms towards important features in the environment that are related to the the agent's goals and plans, but visual perception has traditionally been a distinct area of research with vision systems acting independently of planning.

The goal of visual perception is to accept a visual scene as input and to label the objects and perhaps to identify the relations between objects as output (see for example [57]). Once an agent receives the output of vision, it can search for objects or relationships that bear on goals and plans. This division of labor is quite inefficient since many of the objects in a visual scene will likely never affect the agent. In contrast, an active approach to vision asserts that the vision system should operate with the goals and plans as a guide (c.f.,[30, 32]). In the research reported here, we demonstrate a novel integration of planning and perception that uses goals and plans as a rationale to bias an active visual perception component in a cognitive agent.

We introduce the concept of a monitor as a solution for interleaving these cognitive processes of perception, plan and interpretation. Plan monitors focus perception of aspects of the world relevant to the plan, whereas goal monitors relate to goal related aspects. We assert that an autonomous agent equipped with such monitors can be more robust in a

dynamic world and react to unexpected changes.

First, we focus on the problem of dynamically adjusting planning search using perception during plan generation. Plan monitors focus perception to track the states that form the basis of planning choices. When a feature being monitored changes, the planner will update the plan and alter the planning search instead of resetting the plan generation process from scratch. We will demonstrate in simulated domains that it takes less planning cost when an agent adjusts the planning search by taking into account visually detected changes [27].

We then examine the problem of monitoring goals. If an agent formulates a goal on its own, it should have reasons for doing so. These reasons establish the means for monitoring if the goal is still worth achieving. Our research introduces a cognitive mechanism to anticipate changes in the environment that allows the dropping or changing of a goal and the associated plan to achieve that goal. When the justification for a goal (as opposed to the goal itself) ceases to hold, the agent should also reconsider its goal. We propose goal monitoring as a cognitive process that oversees the continuing benefit of selective goal expressions and when situations warrant, decides whether to abandon or change goals.

We define three types of goal monitors: *operator-based*, *explanation-based* and *direct* goal monitors. Operator-based goal monitors consist of a set of goal operators as rules that generate goals when their conditions are satisfied in the world. This class of goal monitors observe these conditions. Explanation-based goal monitors are created from an explanation used to formulate a goal in response to a discrepancy between the agent's expectations and observations. Explanation-based goal monitors extract the reasons why the agent pursues a goal from an explanation of a discrepancy and perform monitoring of these reasons. Direct goal monitors check to see if the goal is achieved exogenously.

1.1 Contributions

The contributions of this thesis are the following:

- A novel approach integrating planning and perception using plan and goal monitors. Specifically, we describe a modified hierarchical task network planning algorithm that dynamically adjusts its planning search using new information received by visual components (Chapter 3)

- A goal monitoring approach that uses the causes for which the agent pursues a goal and performs monitoring of each cause to ensure the goal is still relevant. First, we describe goal monitors in an operator-based that can be defined by a user. Second, we describe explanation-based that are generated automatically from explanations of a discrepancy (Chapter 4).

- Novel algorithms which create explanation-based and operator-based goal monitors to observe the causal structures of a formulated goal and change or drop the goal during the execution if the agent's beliefs change (Chapter 4).

- Explanation-based goal monitors capable of handling multiple explanations in situations where more than one hypothesis may be true. In partially observable domains, the agent might consider multiple hypotheses. If two or more possible explanations exist, each could be likely initially. But if relevant new information is observed, the system should consider re-ordering the explanations. Goal monitors are created for all possible hypotheses and when the agent observes new information one or more of the goal monitors may fire which in turn will lead to a possible change in the goals the agent is pursuing (Section 4.2).

In this dissertation, we use experimental evidence to defend three claims about the quality of the solutions found by the proposed techniques:

Claim1

There are measurable advantages in integrating interpretation, planning, action and perception in a cognitive architecture. We show that these cognitive processes are dependent. For example, vision algorithms tend to take an image and label them independently of any goal and plan. We show that such algorithms are insufficient. We make perception focused with respect to the agent's behavior and goals, and we show there is cost saving where cost is the number of objects and relations processed.

Claim2

Plan monitors allow the agent to anticipate failures with the plan in a dynamic world and guarantee that a valid plan is produced when plan generation terminates. This method improves the planning performance of an agent in complex domains.

Claim3

Goal monitors improve the goal achievement performance of a cognitive agent in a partially-observable, dynamic environment. This approach provides the basis to change the goal when the goal is no longer useful in the world.

1.2 Outline of the Thesis

This thesis focuses on improving the robustness of a cognitive agent in a dynamic world using novel anticipatory monitors. Chapter 2 starts with the description of the Metacognitive Integrated Dual Cycle Architecture (MIDCA)[1]. MIDCA consists of "action-perception" cycles at both the cognitive level and the metacognitive level. In general, a cycle performs

[1] While we use MIDCA in our experiment, these approaches are relevant to other cognitive architecture in general.

problem-solving to achieve the current goals and then tries to comprehend the resulting actions (and the actions of other agents). The problem-solving component of each cycle consists of intention, planning, and action execution, whereas the comprehension component consists of perception, interpretation, and goal evaluation. There are four key processes in the MIDCA cognitive architecture that are the focus of this thesis: Plan, Act, Perceive and Interpret. Each processes is described in Chapter 2. The interaction between these phases allows us to monitor the plan and goals.

The plan monitor and its algorithm are discussed in Chapter 3. Plan monitors provide a means of focusing visual attention on features of the world likely to affect the plan. First, we discuss how state change can affect the planning decisions. Next, we discuss how to apply a rationale-based monitor technique to a hierarchical planner. We modified a planner to generate plan monitors to interact with a perceptual system and react to environmental changes that bear on planning decisions. We then discuss how the plan monitors respond to these changes. An application of these monitors is tested in a Baxter humanoid robot.

Algorithms for goal monitors are given in Chapter 4. We discuss three types of goal monitors, e.g., operator-style, explanation-based and direct. There are two components to each monitor: the triggering conditions and the associated response. Each component is discussed for each type of monitor.

Chapter 5 provides the evaluation of plan monitors. We evaluated plan monitors in three different domains: blocksworld, logistics, and a physical robot domain. Chapter 6 provides the evaluation of goal monitors. We evaluated the goal monitors in two domains: logistics and Minecraft. These two chapters provide the descriptions of these domains. Empirical results follow demonstrating the benefits of our proposed approaches.

Chapter 7 contains a literature review of the state of art in the following topics: goal reasoning agents, which are able to reason and adjust their goals; cognitive agents, that integrate act, planning, and perception such as SOAR and ICARUS; planning systems, which describe agents that generate a plan to achieve goals and integrate planning and

execution to increase the robustness.

Chapter 8 concludes the thesis. This dissertation demonstrates that the anticipatory monitors are a viable approach for more robust agents in highly dynamic worlds. We summarize the status of our claims, and we list multiple avenues for future research.

Action, Planning, Perception and Interpretation in a Cognitive Architecture

In this dissertation, we build upon the Metacognitive Integrated Dual-Cycle Architecture (MIDCA). While we use MIDCA in our experiment, these approaches are relevant to other cognitive architecture in general. MIDCA is a modular system that has some built-in features like goal reasoning model. Each module in MIDCA can be implemented using different algorithms and is not limited to a single approach. For example, MIDCA uses different kinds of external planners like FF planner, SHOP planner, etc. In this section, we describe the basic concepts of this cognitive architecture and different phases which are the focus of this thesis.

2.1 MIDCA: The Metacognitive Integrated Dual-Cycle Architecture

The *metacognitive, integrated dual-cycle architecture (MIDCA)* [21, 64][1] consists of "action-perception" cycles at both the cognitive and the metacognitive levels (see Figure 2.1). In general, a cycle performs problem solving to achieve its goals and tries to comprehend the resulting actions and those of other agents and events. The problem-solving mechanism of each cycle consists of intention, planning, and action execution processes, whereas the comprehension mechanism consists of perception, interpretation, and goal evaluation. Each process is represented as a phase in a cycle. In this dissertation, we focus on the cognitive rather than the metacognitive cycle.

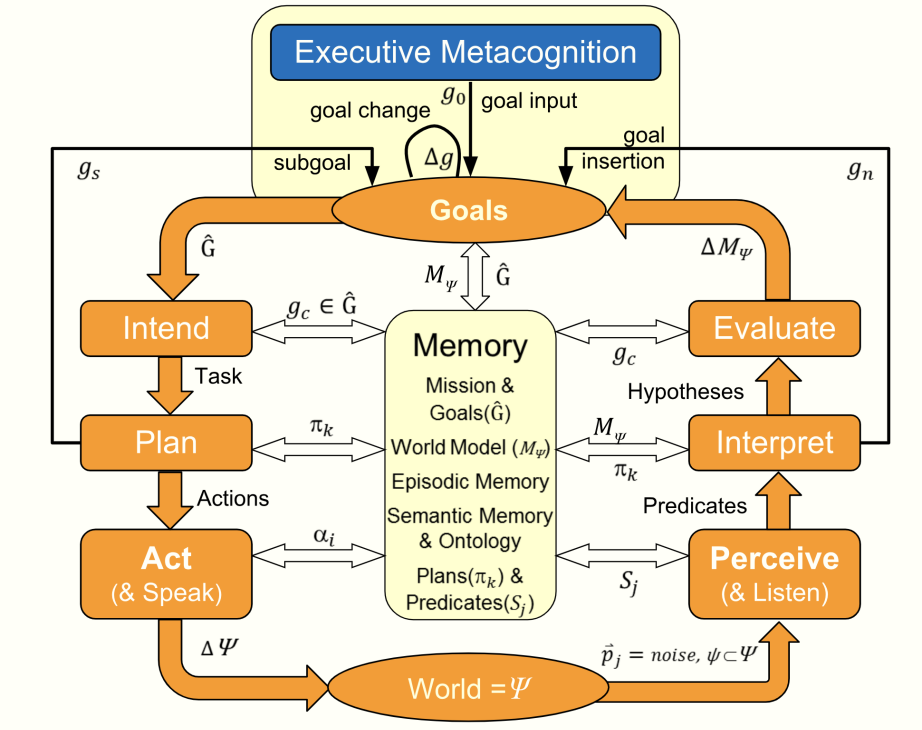

Figure 2.1: The MIDCA architecture. Together intend, plan, and act compose the problem-solving mechanism in the architecture, and perceive, interpret, and evaluation constitute the comprehension mechanism.

[1] https://github.com/COLAB2/midca

In problem solving, the *intend* phase selects and commits to a current goal g_c from those available in its goal agenda $\hat{G}$ (see [49] for details of this phase). The *plan* phase then generates a plan representation π composed of a sequence of steps or executable actions α_i. The plan is executed one step at a time by the *act* phase to change the world through the effects of the planned actions. The agent will then use the plan as expectations in the next cycle to evaluate the execution of each step in the plan.

Comprehension starts with perception of the world Ψ mapping percepts $\vec{p}$ to symbolic predicate representations S_j via the *perceive* phase. The *interpret* phase takes as input the resulting predicates and the expectations in memory to build a representation M_ψ of the sequence of actions and any exogenous events that co-occur. Interpret determines whether the agent is making sufficient progress toward its goals. The *evaluate* phase determines whether the current goal g_c is actually achieved in the world and removes it from memory if true. This cycle of problem-solving and action followed by perception and comprehension functions over discrete state and event representations of the environment.

2.1.1 MIDCA Environments

MIDCA version 1.4 is the latest implementation of the MIDCA architecture whose components are shown in the Figure 2.1. MIDCA_1.4 adds an application programming interface (API) to communicate with agents in either a standard planning simulator or the *Robot Operating System* (ROS) [66] (see Figure 2.2).

Standard Planning Simulator

MIDCA uses an explicit model of the environment. To simulate the actions executed by the act phase, we use the MIDCA's world simulator. We used three different domains in the experiments in this dissertation: blocksworld, logistics, and minecraft[2] (described

[2]http://www.minecraft.net/

in Sections 5.1 and 6.1). The blocksworld domain includes a set of blocks sitting on a table. The goal is to build one or more vertical stacks of blocks. The logistics domain [72] represents a simplified shipping problem involving planes, trucks, cities, and airports. Goals generally specify the destination of packages. In the minecraft[3] domain, a character named Steve explores a simulated finite 2D world while gathering resources and surviving dangers. For example, the goal can be gathering seven pieces of wood to build a pickaxe.

2.1.2 The ROS MIDCA Interface

ROS is a distributed publish/subscribe middleware that enables command and control functions for sensor and effector platforms. Through subscribed ROS topics and executable nodes, MIDCA can control a Baxter humanoid robot in the real world and in the Gazebo simulator. A ROS node is an executable computation that communicates with other nodes via communication paths represented as topics. We created ROS nodes that are responsible for doing specific actions such as moving the Baxter's arms and additional nodes for getting object spatial representations. The API specifies the types of and meanings for incoming and outgoing messages on various ROS topics so that MIDCA and the robot can interact effectively. When a robot action executes, the API places feedback messages in a MIDCA memory buffer that indicate success or failure states of the effector. As asynchronous messages from the robot sensors occur, interface handlers place them in additional buffers for perceptual processing. During the perceive phase (see again Figure 2.1), MIDCA will access the messages, manipulate the content, and store the results.

[3]The real game is an infinite 3D world. We use a simulated finite 2D world.

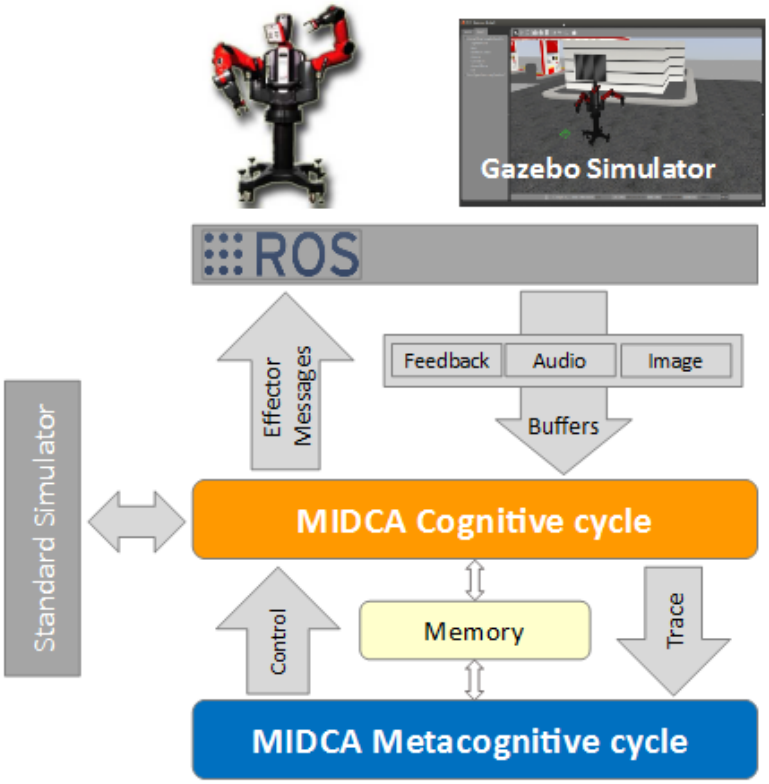

Figure 2.2: The block diagram of interfaces for MIDCA, version 1.4 [21]. MIDCA communicates with ROS and a Baxter humanoid robot through the API.

2.2 The Act Phase: Mapping action models to agent behavior

MIDCA chooses the next action from the current plan, if one exists. Otherwise, it does not perform an action. If an action is chosen, it is sent to the world simulator to compute the next world state.

Act Phase in the ROS MIDCA Interface

The plan phase creates a high level plan by using a planner, then transforms it into an actionable plan using a mapping between high-level actions and methods to carry them out. The act phase operationalizes the plan step using a mapping between high-level actions and

directly executable methods for the robot. For example, the plan step *moveto(loc(object))* (*moveto* is an operator to move the robot's arm to the location of an object) is instantiated in a method that sends out a ROS message containing location coordinates to the node that operates the arm, then repeatedly checks for feedback indicating success or failure. In the act phase, actions are begun successively until either one fails or a blocking action is reached. Once all actions in a plan are complete, the plan itself is considered complete. If any action fails, the plan is considered failed.

2.3 The Plan Phase: Generating a sequence of actions from the goal

2.3.1 Problem Representation

Planning is concerned about finding a sequence of actions that translate the initial state to the goal state. The classical representation scheme generalizes the set-theoretic representation scheme using notation derived from first-order logic. States are represented as sets of logical atoms that are true or false within some interpretation. Actions are represented by planning operators that change the truth values of these atoms.

States

A state s is a finite set of ground atoms of a first-order language $\mathcal{L}$. Goals are represented as subsets of states. We say that the agent has achieved its goal g when the current state s satisfies g (i.e., $s \models g$).

Example 1. Suppose we want to formulate a blocksworld planning domain (described in Section 5.1.1) in which there are three blocks (R, B, and G). One of the states is the state

s_0 and is shown in Figure 2.3.

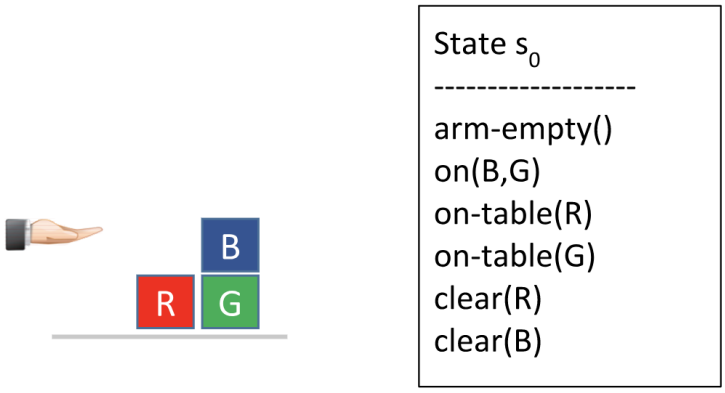

Figure 2.3: The blocksworld state s_0.

Operators and Actions

A planning operator is a triple o = *(name(o), precond(o), effects(o))* whose elements are as follows: *name(o)*, the name of the operator, *precond(o)*, the preconditions of the operator and *effects(o)*, the effects of the operator. For an operator to be applicable in a given state, the preconditions must be valid in the state. The effects of an operator represent the changes to the state after the operator has been executed.

An action is any ground instance of a planning operator. If $\alpha \in A$ is an action and $s \in S$ is a state such that $\textit{precond}^+(\alpha) \subseteq s$ *and* $\textit{precond}^-(\alpha) \cap s = \emptyset$, then α is applicable to s, and the result of applying α to s is the state: $\gamma(s, \alpha) = (s - \textit{effects}^-(\alpha)) \cup \textit{effects}^+(\alpha)$. *Add-list* is the list of atoms that are added to the state, and *delete-list* is the list of atoms that will be removed from the state after executing an action.

Example 2. Here are some planning operators for the blocksworld domain (see the Appendix A for the complete domain description):

name: pickup(a)

precond:clear(a), on-table(a), arm-empty()

effects: holding(a), ¬ on-table(a)

———————-

name: unstack(a,b)

precond:clear(a), on(a,b), arm-empty()

effects: holding(a), ¬ on(a,b)

———————

name: stack(a, b)

precond: clear(b), holding(a)

effects: ¬ holding(a), on(a, b), ¬ clear(b), arm-empty()

Example 3. Action unstack(B, G) is a ground instance of the operator unstack(a,b) which is applicable to state s_0 of Figure 2.3. The result is the state $s_1 = \gamma(s_0, \text{unstack } (B))$ shown in Figure 2.4.

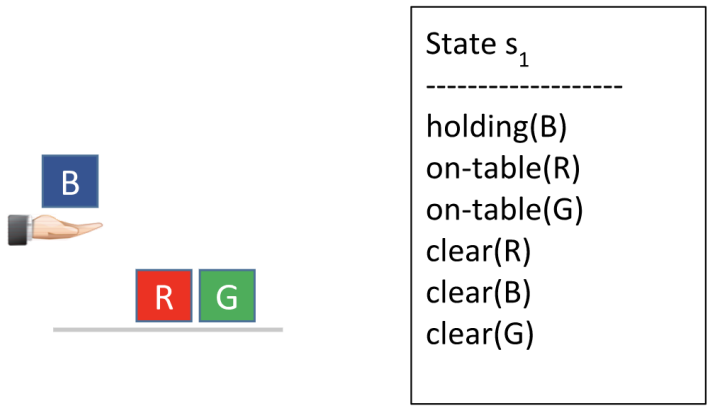

Figure 2.4: The blocksworld state s_1.

Problems and solutions

A classical planning domain is defined as a finite state-transition system in which each state $s \in S$ is a finite set of ground atoms of a first-order language $\mathcal{L}$. For a classical planning domain, the state-transition system is a 3-tuple $\Sigma = (S, A, \gamma)$, where S is the set of all states, and A is the set of all actions. In addition, gamma is a state transition function $\gamma : S \times A \rightarrow S$ that returns the resulting state of an executable action given a current state.

A classical planning problem for a restricted state-transition system Σ is defined as a triple $\mathcal{P}$ = (Σ, s_0, g), where s_0 is an initial state and g corresponds to a set of goal states. A solution to $\mathcal{P}$ is a sequence of actions $\pi = \langle a_l, a_2, ..., a_k \rangle$ corresponding to a sequence of state transitions (s_0, s_l, ..., s_k) such that $s_1 = \gamma(s_0, a_l)$,..., $s_k = \gamma(s_{k-l}, a_k)$, and s_k is a goal state such that $s_k \models g$.

Example 4. Consider the following plan: π_1 = $\langle$ unstack(B, R), stack(B, G) $\rangle$. This plan is applicable to the state s_0 shown in Figure 2.3, producing the state $s_g = \{$on(B,G), on-table(R), on-table(G), arm-empty(), clear(B), clear(R)$\}$. π_1 is a solution to the blocksworld problem $\mathcal{P}$ whose initial state is s_0 and whose goal g is *on*(B, G).

Events

An event template is defined the same as a classical planning operator: e=(name(e), precond(e), effects(e)) where *name* is the name of the event, *preconds* and *effects* are the preconditions and effects of the event respectively. An event occurs when all of its preconditions are met in the true world state. Since actions and events change the current state of the world, events can be triggered at any time.

Occurrence o refers to the occurrence of any observation obs, action a, or event e. An execution history is a finite sequence of observations and actions $\langle obs_0, a_1, obs_1, a_2, ..., a_k,$ $obs_{k+1} \rangle$.

2.3.2 Planners in MIDCA

In this dissertation, we used the SHOP[4] planner [62] and the metric-FF [41] planner. SHOP[5] is an Hierarchical Task Network (HTN) planning algorithm that creates plans by recursively decomposing tasks into smaller subtasks until only the primitive tasks are left which can be accomplished directly [62]. Metric-FF[6] is a forward chaining heuristic state space planner. Metric-FF is an extension of the FF planner to numerical state variables. We used Metric-FF for experiments in the minecraft domain (described in section 6.1.2). The MIDCA agents operate in planning domains using the Planning Domain Definition Language (PDDL) version 2.1 [33] representing actions and predicates.

Hierarchical Task Network (HTN) Planning

In an HTN planner, the objective is to perform some set of tasks instead of achieving a set of goals. The input to the planning system includes a set of operators and a set of methods, each of which is a prescription for how to decompose some task into some set of smaller tasks. HTN planning decomposes nonprimitive tasks recursively into smaller subtasks, until primitive tasks are reached that can be performed directly using the planning operators.

An HTN planning problem is a 3-tuple $\mathcal{P} = (s, T, D)$. It takes the initial state, s, which is a symbolic representation of the state of world, and a set of tasks, T=$\langle t_1,...,t_k \rangle$, to be accomplished. Also, it takes a knowledge base, D, including operators and methods. A plan π = $\langle \alpha_1, ..., \alpha_n \rangle$ is a solution for a planning problem to accomplish T. This means that there is a way to decompose T into π in such a way that π is executable in s, and upon execution will transform the start state into the goal state.

We denote an action α=*(name(α), precond(α), effects(α))* that accomplishes a primitive task t in state s if *name(α)* = t and α is applicable to s. A method is a 4-tuple m

[4]The version we use is JSHOP(the SHOP planner written in JAVA).
[5]http://www.cs.umd.edu/projects/shop/description.html
[6]https://fai.cs.uni-saarland.de/hoffmann/metric-ff.html

= *(name(m), task(m), precond(m), subtasks(m))* in which *name(m)* is the name of the method; *task(m)* is a non-primitive task; and *precond(m)* is a set of literals called the method's preconditions. *Precond(m)* specifies what conditions the current state must satisfy in order for m to be applied, and *subtasks(m)* specifies the subtasks to perform in order to accomplish *task(m)*.

Example 5. Figure 2.5 shows an example of a Baxter robot with the goal *on(G(reen_block), R(ed_block))*[7]. This goal maps to a root task, $move\_blocks$, in the SHOP planner. The planner decomposes $move\_blocks$ to the non-primitive ⟨ *pickupT(G),stackT(G,R)* ⟩ tasks, in that order. Task *pickupT(G)* decomposes to the primitive tasks ⟨ *moveto(loc(G)), grasp(G)* ⟩, and *stackT(G,R)* decomposes to the primitive tasks ⟨ *moveto(loc(R)),stack(G,R)*⟩, which $Loc(R)$ is an x,y,z coordinate specifying the location of the red block.

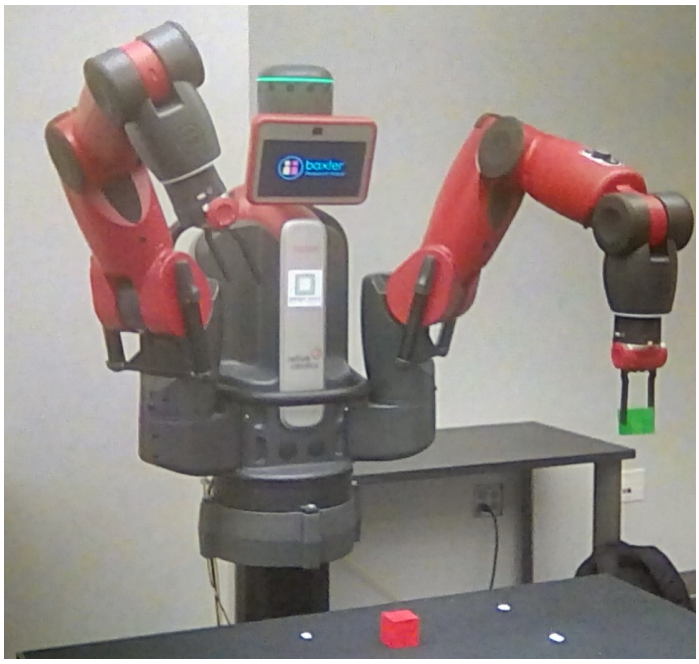

Figure 2.5: The Baxter is stacking a green block on a red block. Baxter's right hand's camera is used for receiving images. The left arm is used for executing actions.

[7]We use a different blocksworld planning domain for Baxter. See Appendix B.

The Fast Forward (FF) Planner

The *Fast Forward (FF)* planner [43] performs forward search (a variation of hill-climbing) in the space of all reachable states, and heuristic evaluation is done by means of solving a relaxed task (a task is relaxed by ignoring all delete lists) in each single search state, using a Graphplan-style [9] algorithm (i.e., to solve the relaxed tasks). The resulting relaxed plans inform the search by means of a goal distance estimation (the number of actions) as well as by means of an estimation of which actions are most useful.

2.4 The Perceive phase: Creating a symbolic world from visual images

The perceive phase generates discrete world states which are represented symbolically as logical predicates on objects in an image. In the perceive phase, MIDCA reads messages from all the buffers, processes them and stores the processed data in MIDCA's main memory. As such, the perceive phase is responsible for reasoning about vision messages and creating world states which are represented symbolically as logical predicates. Although outside of the scope of this dissertation, the API handles audio through a speech-to-text library and places character strings in the audio buffer when utterances are spoken. Here we concentrate on vision and the image buffer instead.

Figure 2.6 shows the two stages of visual detection and predicate extraction to create the symbolic world from a given image. As the Baxter's camera reads in images, a ROS node running concurrently with MIDCA performs a *visual detection* procedure to locate the objects and sends entity data (e.g., color, location) about any known objects to the buffer. The API pairs coded object labels (e.g., red-block) with their location coordinates in the image and places them in the image buffer of the API. The object detection algorithm uses

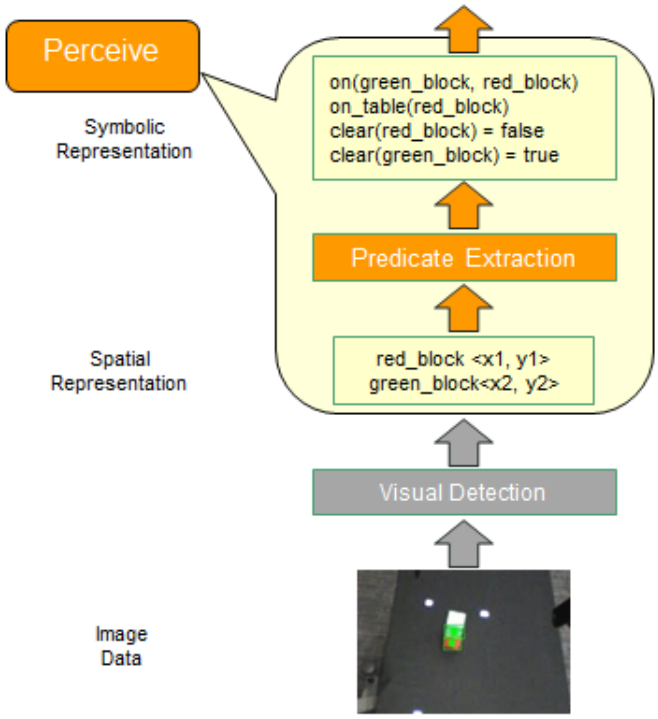

Figure 2.6: From image to symbolic world in two stages (details of perceive phase from Figure 2.1). Given image data from the Baxter's right hand camera, visual detection inserts spatial representations into the API image buffer. Predicate extraction then transforms the spatial information into symbolic relations between objects in the world.

functions from the OpenCV library (opencv.org) to detect objects using their color.

In the second stage, the perceive phase performs *predicate extraction*. This procedure reads the spatial representation from the buffers and extracts predicates using the distance between objects. For example, in Figure 2.6, first the blocks are detected in the image(red block and green block), and then the predicates like *on(green-block, red-block)* are extracted using the distance between the blocks (shown in the second box). Perceive stores both spatial and symbolic representations in MIDCA's memory.

2.5 The Interpret Phase: Goal reasoning

The interpret phase has been at the core of this research. MIDCA uses two approaches to analyze the current state of the world and determine if any new goals needs to be pursued (see Figure 2.7). In the first approach, the goal is formulated by goal operators. When the conditions of these operators are satisfied in the current state a goal is formulated. Other cognitive architectures like Soar [50] take this approach. Operators exist for various goal types and data-driven context-sensitive rules spawn them given matching run-time observations.

The second approach is implemented based on a type of goal reasoning called *Goal Driven Autonomy (GDA)* [3, 20, 48]. In this approach, the goal is formulated in response to a discrepancy between the agents expectations and the observation. An explanation provides the antecedents for the discrepancy, and the agent generates a goal from the explanation.

In the interpret phase, MIDCA performs many functions that attempt to make sense of the input given what the agent is trying to achieve (i.e., its goals) and what plan steps have been executed. The GDA approach to robust autonomy asserts that goal management is central to handling complexity in dynamic environments under limited resources.

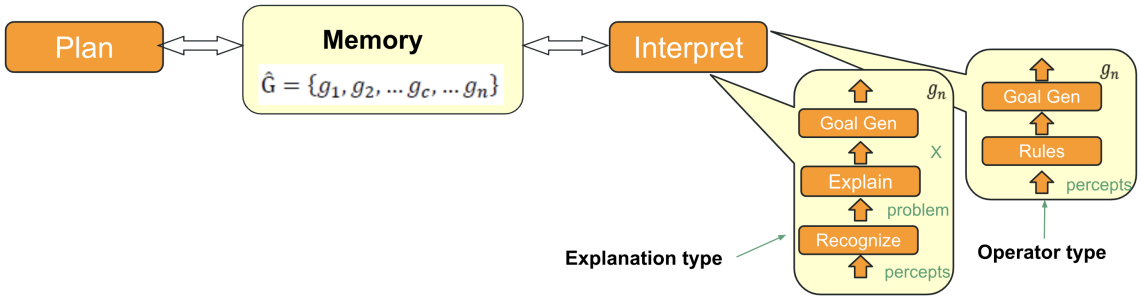

Figure 2.7: Two Goal formulation approaches in MIDCA: (1) Goal is formulated based on expert authored rules. (2) Goal is formulated from an explanation of a discrepancy. X stands for explanation.

2.5.1 Goal Reasoning

Recent work on goal reasoning [1, 2, 40, 68] has started to examine how intelligent agents can reason about and generate their own goals instead of always depending upon a human user directly. Broadly construed, the topic concerns complex systems that self-manage their desired goal states [71]. In the decision-making process, goals are not simply given as input from a human, rather they constitute representations that the system itself formulates.

Goal-driven Autonomy

Cox's INTRO [19] system integrated planning, execution, and goal reasoning which provides inspiration for several concepts in GDA. Aha et el. [3] extended this idea and integrated it with Nau's [63] online planning framework. This work first introduced the GDA model. GDA is a kind of goal reasoning that focuses on explanation of discrepancies to formulate new goals. GDA agents generate goals as the agent encounters differences between the agents expectations for the outcome of its actions and the actual observed outcomes in each new state [26]. When such a discrepancy occurs, GDA agents generate an explanation for the discrepancy, and generate new goal(s). A diagram of a GDA agent is shown in Figure 2.8. The GDA model performs the following four steps:

1. *Discrepancy detection* compares observations with its expectations and generates a set of discrepancies D,

2. *Explanation generation* hypothesizes one or more explanations E for D,

3. *Goal formulation* generates zero or more goals taking into account explanation E,

4. *Goal management* selects which goal to pursue next.

The four step GDA process is shown within the Controller which forms the core of the GDA model. The planner takes a model of the environment, a current state and a goal to

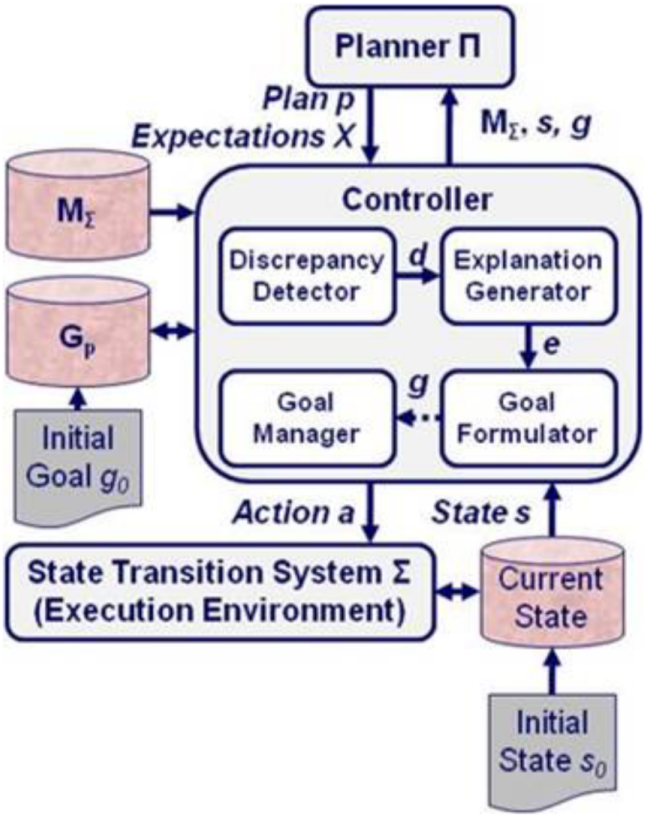

Figure 2.8: A basic model of a Goal Driven Autonomy agent [60].

generate a sequence of actions to achieve the goal. Also, GDA's planner generates a set of expectations. The controller uses the plan to apply an action to the state transition system to update the current state.

Example 6. In a modified blocksworld domain [65], a hidden arsonist is added, who can set blocks on fire. Consider a scenario in this domain where a block catches on fire in the middle of the plan execution. MIDCA's interpret finds this anomalous and uses an explanation module to explain this anomaly. Paisner [65] used Meta-AQUA in the interpret phase of MIDCA [24] to generate explanations. Meta-AQUA explains this anomaly by hypothesizing that the burning was caused by an arsonist. A goal g' to apprehend the arsonist is generated. MIDCA's intend process suspends the current goal to achieve the apprehend goal g' first. Upon completion of g', MIDCA goes back to its original goal. Note that the anomaly detection and explanation generation are not the focus of this research. We

only give a short description for these steps. For more details, see [26, 19, 61].

Discrepancy Detection and Explanation Generation

We used DiscoverHistory [61, 59] to formulate explanation when there is a discrepancy. We used DiscoverHistory in MIDCA instead of Meta-AQUA because DiscoverHistory uses the same planning domain (PDDL) as the planner, so it needs less domain engineering. Meta-AQUA's need for explanation patterns scales with the number of explanation types supported, whereas DiscoverHistory's action and event models scale to the environment itself. DiscoverHistory will therefore require less knowledge in complex environments with many kinds of possible explanations. This explanation module is used in the interpret phase in MIDCA to reason about the causes of inconsistencies between observations and expectations that arise during plan execution and generate abductive explanations. In the next section, we give a brief description of DiscoverHistory.

DiscoverHistory

DiscoverHistory [61] abductively reasons about unexpected events that occur during plan execution. DiscoverHistory hypothesizes what events occurred to cause the discrepancy and provides the causal structure that caused the agent's plan to fail. We take the same definition for explanation as [61]: the planning agent's knowledge about the changes in its environment is an explanation of the world. DiscoverHistory constructs explanations by inferring root causes of an anomaly that were not observed.

The interpret phase in MIDCA passes the execution history to DiscoverHistory, and DiscoverHistory generates successive explanations by attempting to resolve inconsistencies. An *execution history* is a finite sequence of observations and actions $\langle\, obs_0, a_1, obs_1, a_2, ..., a_k, obs_{k+1}\rangle$. If the agent's expectations for the outcome of its actions are different from the actual observed outcomes in each new state, we say there is an inconsistency. We

investigate only inconsistencies between an observation and preceding actions. Such an inconsistency is a tuple (p, o, o'), such that p is a proposition in the new observation o' (o' is an observation and $p \in obs$) but p is not an effect of the previous occurrence o (o is an action a and $\neg p \in$ effects(a)).

In response to the anomalies, new goals are formulated to remove the cause of each anomaly as identified by the explanations. Goal monitors observe the causal information of the explanations to make sure the goal remains relevant (Chapter 4). The planner then generates a plan to achieve the goal and plan monitors react to environmental changes that are relevant to the plan (Chapter 3). When either monitor type detects such changes, they execute specific plan or goal modifications as needed.

Monitoring the Dynamic Environments: Plan Monitors

In practical planning situations, the planning activity may take a few hours, days or weeks (for example, the military planning for Desert Storm took five months [15]). Relevant information is thus likely to change during the planning process. These changes may affect planning decisions and need to be monitored so that the plan can be refined, otherwise the resulting plan is no longer valid [6, 73]. Our approach lets the planner have access to information from the environment and enables perception to be sensitive to the agent's plans.

Consider a robot that picks and places objects on a conveyor belt or table surface. In Figure 3.1, we see a robot that grasps a green block before it attempts to place it on a red block. But if a person or any other agent suddenly places a third object on the red one, its plan will fail. Now if the planning, acting, perception, and interpretation processes interact properly, like any human, the robot should be able to adjust its plans, goals, and behavior seamlessly. This should be the case both during planning and during plan execution (here we will focus on the former). Unfortunately, the predominant scientific research efforts on planning, goal reasoning, and perception have been investigated separately.

In the following sections, we provide an overview of our proposed approach to solve this problem. We describe the algorithm that creates the perceptual plan monitors to inte-

grate the planning and interpretation. We then discuss how these monitors are added to the MIDCA cognitive architecture. Finally, we describe a demonstration of a humaniod Baxter robot with MIDCA to show how the plan monitors work in the actual world.

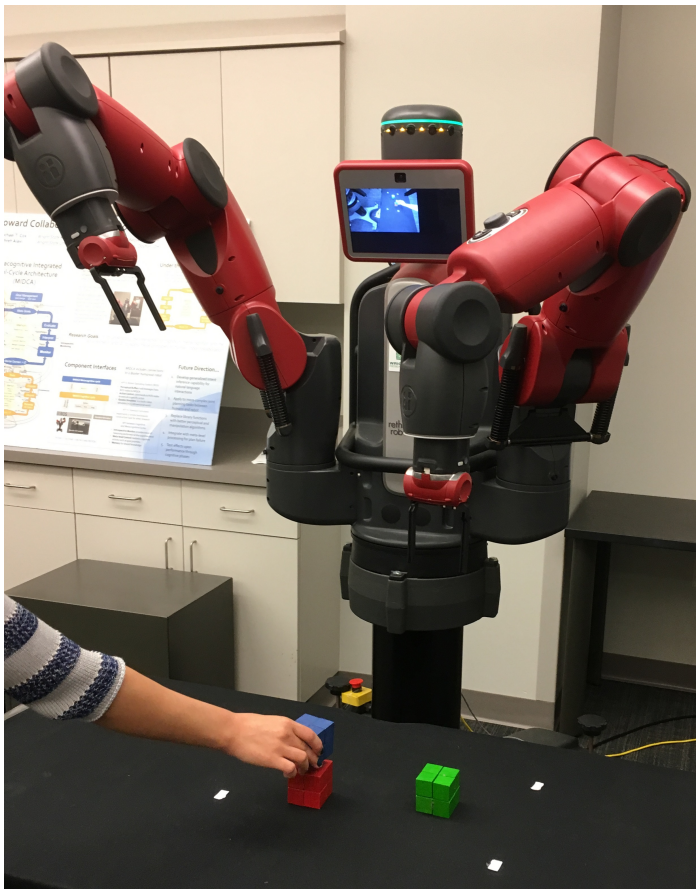

Figure 3.1: A student puts the blue block on the red block during planning for goal $on(R, G)$. A video of MIDCA controlling a Baxter robot is available at https://tinyurl.com/y3z2e98r

.

3.1 Perceptual Plan Monitors

A *perceptual plan monitor* provides a means of focusing visual attention on features of the world relevant to what the agent is trying to do. We claim that interpret, plan, act and perceive should interact with each other and that a perceptual monitor represents the

construct for this purpose. Perceptual plan monitors observe the conditions of operators in the plan under construction. When observations are different from the agent's expectations as specified in the operator, they trigger plan transformation.

MIDCA creates plan monitors as it adds each step to the plan during the Plan phase. Figure 3.2 shows plan monitors in the MIDCA architecture. These monitors call perceive to observe the relevant features in the state. When a feature being monitored changes, and the change is detected, we say that the monitor *fires*. Deliberation can then be performed to decide whether the plan under construction should be changed. If the planner decides to account for the new change, it will update the plan and alter the planning search. In particular, parts of the plan may be deleted because they have become unnecessary; new tasks may be added and current ones refined; and prior decisions about how to achieve particular goals may be revisited [73]. We use these monitors in the SHOP planner to make perception focused on the relevant features to the agent's plan.

A change in the world can result in different kinds of plan transformations. Veloso, Pollack and Cox [73] organized these transformations into three different categories:

1. Extending the plan with additional actions;
2. Shortening the plan by removing actions;
3. Substituting a different plan with alternative bindings or steps.

This paper focuses on the first two types. See [73] for a discussion of the third type of transformation.

3.2 The Influence of State in Planning Decisions

Planning decisions are influenced by MIDCA's beliefs about the state of the world and the goals. Given a task t, MIDCA repeatedly needs to decide how to decompose the task to

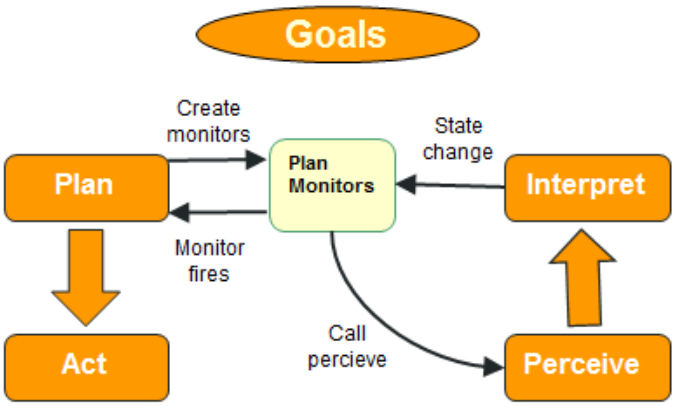

Figure 3.2: Conceptual representation of perceptual plan monitors. The plan phase generates the plan monitors after an operator is added to the plan to watch its conditions. The monitors call perceive. Interpret checks if the perceived information is as the same as the expectations. If not, the plan phase refines the plan.

achieve t until the decomposition reaches primitive tasks represented as operators. The decision to pursue a decomposition over another one will depend on the current state of the world. When a satisfied precondition of an operator becomes unsatisfied during planning, the planner needs to add steps to the plan to reestablish the condition. In other situations, when a portion of the current plan serves to establish some condition c, it may become necessary to cut those actions from the plan, should c become true spontaneously.

For example, assume the initial state in panel (a) of Figure 3.3 with the three blocks R, G, and B on the table. The goal g is *on (R,G)*, and the task to accomplish g is *stack-T(R,G)*. Given that both blocks are clear, the planner generates a simple two-step plan π = $\langle pickup(R), stack(R, G)\rangle$. Panels (b) and (c) show the execution of the plan steps.

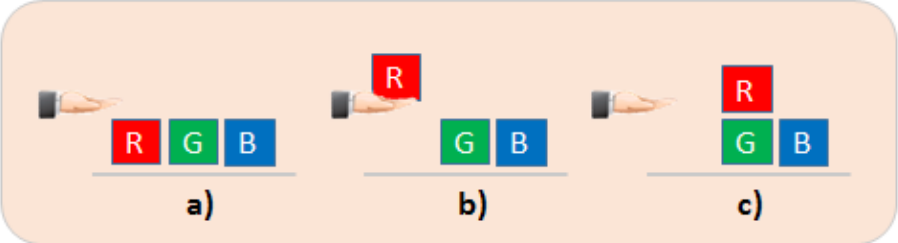

Figure 3.3: A blocksworld problem to put block R on G. The first panel shows the initial state, and the remaining panels show the incremental execution of the plan steps that solve the problem.

Now consider the situation whereby another agent puts the block B on top of R while MIDCA is planning. This new state violates the *clear(R)* precondition of the *pickup* operator (see Appendix A for definition of pickup operator). Therefore, the planner must add further actions to the plan before continuing. The refined plan π' to achieve goal g is $\langle$pickup(B), putdown(B), pickup(R), stack(R,G) $\rangle$.

3.3 Perceptual Monitors in SHOP

In this section, we explain how we modified the SHOP planner to adapt search in response to changes during planning. Algorithm 1 shows the overall procedure. The modified SHOP algorithm takes the initial state, s, a set of ordered tasks, $\langle\ t_1,...,t_k\ \rangle$, and a knowledge base, D, including operators and methods. We added another argument, l, to the planner to keep track of the recursion tree depth. A plan π= $\langle\ \alpha_1,...,\alpha_m\rangle$ is the solution of this algorithm. Note that the symbol . at line 27 refers to concatenating α to π. To integrate with rationale-based monitors, two procedures are added to the SHOP planner (steps 3-15 and step 26). First, the monitors are generated when an operator is added to the current plan, π (line 26 in the Algorithm 1). Second, at each planning cycle, the SHOP planner checks for fired monitors. If a monitor fires, the planner refines the plan (steps 3-15 in Algorithm 1).

Algorithm 2 shows the details of monitor generation for the preconditions of an oper-

Algorithm 1 SHOP with Perceptual Plan Monitors. **Input:** World state s, list of tasks $\langle t_1, ..., t_k \rangle$, knowledge base D, recursion tree depth l. **Output:** Plan π.

1: $l \leftarrow 0$ $mnts \leftarrow \langle\rangle$
2: **procedure** $SHOP(s, \langle t_1, ..., t_k \rangle, D, l)$
3: $new\_s, \langle (p_1, l_1), ...(p_n, l_n) \rangle \leftarrow fired(mnts)$
4: **if** $n \neq 0$ **then** ▷ at least one monitor fired
5: $s', T' \leftarrow backtrack(l_1)$ ▷ s' and T' are the state and list of tasks at depth l_1
6: $s' \leftarrow$ update s' with new_s ▷ state changed
7: **while** $T' \neq \phi$ **do**
8: $t' \leftarrow first(T')$
9: **if** t' is primitive or $precond\ (method(t')) \not\models s'$ **then**
10: $l_1 = l_1 - 1$ $s', T' \leftarrow backtrack(l_1)$
11: $s' \leftarrow$ update s' with new_s
12: **if** $T' \neq \phi$ **then**
13: return $SHOP(s', T', D, l_1 + 1)$
14: **else**
15: return $\langle\rangle$
16: **if** $k = 0$ **then** return $\langle\rangle$ ▷ if list of task is empty
17: **if** t_1 is primitive **then**
18: $active \leftarrow \{(\alpha, \sigma) | \alpha$ is an operator instance from D, σ is a substitution where α is relevant for $\sigma(t_1)$ and α is applicable to $s\}$
19: **if** $active = \phi$ **then**
20: return failure
21: randomly choose any $(\alpha, \sigma) \in active$
22: $\pi \leftarrow SHOP(\gamma(s, \alpha), \sigma(\langle t_2, ..., t_k \rangle), D, l + 1)$
23: **if** $\pi =$ failure **then**
24: return failure
25: **else**
26: $mnts \leftarrow generate\_mnts(\alpha, l, s, mnts)$
27: return $\alpha.\pi$
28: **else if** t_1 is nonprimitive **then**
29: $active \leftarrow \{(m, \sigma) | m$ is an instance of a method in D, σ is a substitution where m is relevant for $\sigma(t_1)$ and m is applicable to $s\}$
30: **if** $active = \phi$ **then**
31: return failure
32: randomly choose any $(m, \sigma) \in active$
33: $w \leftarrow subtasks(m).\sigma(\langle t_2, ..., t_k \rangle)$
34: return $SHOP(s, w, D, l + 1)$

ator. It takes the operator, o, the current depth, l, state, s, and the list of monitors, $mnts$, as input parameters. Monitors observe features that directly influence π. This includes preconditions of all the operators in π. Some of these preconditions will be true when they are added to π; they therefore must be monitored, because, should they become false, π will fail unless additional planning is performed. Other preconditions will be initially false; should they become true, then the portions of π that established them may become unnecessary (steps 3-6 in Algorithm 2). When monitors are generated, the current recursion depth is recorded and backtracking uses this information later (plan refinement is done by backtracking to the recursion depth that an action fails).

Algorithm 2 Generate plan monitors. **Input:** Plan operator o, the planning recursion depth l, world state s, and a list of plan monitors $mnts$. **Output:** list of plan monitors $mnts$

```
1: procedure generate_monitors(o, l, s, mnts)
2:     for p in precond(o) do
3:         if satisfied(p, s) then
4:             mnts ← (p, l) ∪ mnts
5:         else if ¬ satisfied(p, s) then
6:             mnts ← (¬p, l) ∪ mnts
7:     return mnts
```

Algorithm 3 shows the details of checking for fired monitors. Perceive creates a set of percepts from environmental input (Ψ) and induces a predicate representation s from these percepts (line 4) [4, 28]. Perceive is parametrized to focus only on the desired features. Then it checks to see if the preconditions of the operators in the plan so far are still satisfied in s. If not, it adds the monitor to the list of fired monitors. Then Algorithm 1 uses this information to refine the plan.

Algorithm 3 Check for fired plan monitors. **Input:** list of plan monitors $mnts$. **Output:** list of fired monitors *fired-list*

```
1: procedure fired(mnts)
2:     fired-list ← ⟨⟩
3:     for (p, l) in mnts do
4:         s ← perceive(Ψ,p)
5:         if s ⊭ p then
6:             fired-list ← (p, l) ∪ fired-list
7:     return s, fired-list
```

3.4 Plan Refinement

The core of our approach is how to refine the plan under construction with backtracking and altering the task-decomposition. When an action fails (the preconditions of the operator are not met in the new state), the modified SHOP planner backtracks to the depth that the failed action is added to the plan (steps 4-19 Algorithm 1). The refining procedure starts with traversing the parent links until it finds the closest valid parent of the failed task. A valid parent is a non-primitive task which has been decomposed by a method for which all preconditions of that method are true in the new state (steps 8-16 Algorithm 1).

Examine the task decomposition of t_1 in Figure 3.4. During the planning phase, if task t_{21} fails because the preconditions of operator op_1 become unsatisfied, the SHOP planner will have to backtrack to t_{21} to refine the plan. To check if t_{12} is a valid task, the planner checks if the preconditions of the method m_1 are true, because t_1 is decomposed to t_{12} using method m_1. If there is a precondition of method m_1 that is not satisfied in the currently observed state of the world, then this means t_{12} is also a failed task. The process continues until it finds a task that is valid in the current state, or it reaches the goal task. When the algorithm finds a valid task it tries to decompose it in another way and continues building the rest of the plan.

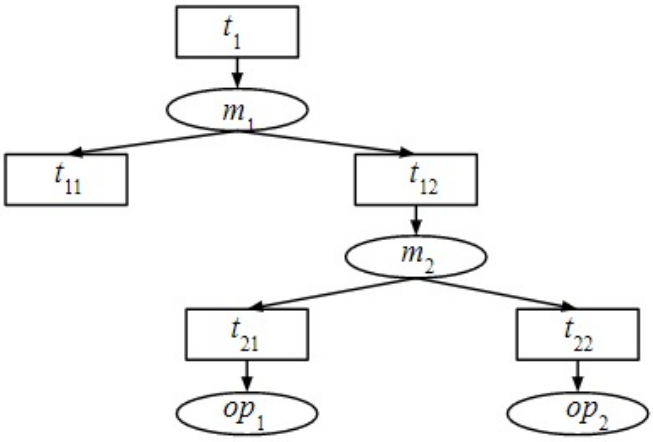

Figure 3.4: An example of task decomposition. Method m_2 refines the non-primitive task t_{12} into t_{21} and t_{22}. Operators op_1 and op_2 accomplish each of these primitive subtasks respectively [28].

3.5 A Perceptual Plan Monitor Demonstration on a Humanoid Robot

We tested our system with a Baxter humanoid robot in a simple blocksworld task to examine the plan monitor implementation in the real world. The implementation interleaves planning, interpretation, and perception in the MIDCA cognitive architecture and demonstrates that plan monitors apply to more complex autonomous systems than simulated worlds alone.

In the plan phase, plan monitors are mapped to a corresponding component in perceive that is only concerned with state changes related to that specific precondition. We used expert authored perceptual functions (the functions that extract the predicates from the image) for perceiving each predicate (e.g., *on* and *clear*) and mapped these to monitors in the planner. The planner checks to see if the observed state is the same as the expected state (various approaches to using expectations to detect discrepancies are described in [26]). If any change happens, the planner refines the plan based on the new state.

In the example from Figure 3.1, the robot plans to achieve the goal *on(R,G)*, and

a student moves the blue block from the red block to the green block in the middle of planning. Depending on the exact point in the planning process that this occurs, perceptual monitors will adapt the plan as appropriate. If the planner already added the action *unstack (B, R)*, the corresponding monitor fires and cuts this step from the plan. If the planner added the step *pickup(G)* to the plan, then two monitors fire and, in addition to cutting *unstack (B, R)* from the plan, it will add *unstack (B)* to the plan. This demonstration shows that MIDCA operates as intended when the world changes during planning.

3.6 Discussion

The integration of planning and interpretation in a cognitive architecture is not a simple one-way interaction. Here, we have argued that perception (particularly vision) should serve the needs of the planner. The planner generates perceptual monitors for the underlying vision system based on the rationale for plan decisions (e.g., preconditions). The interpret phase detects when these conditions are violated. However, it can equally be argued that the planning phase should serve the needs of perception and interpretation. Given a particular scene or situation, MIDCA's interpretation phase can recognize new problems in terms of expectation failures or discrepancies. The interpretation system will then attempt to explain the discrepancy and use the explanation to generate a goal to remove the problem. The goal is passed to the problem-solving module of MIDCA, and the planner will generate a plan to achieve it.

Not only are the plans of the agent important for perception and vision, but also the goals are as well. As the world changes, goals may become out of date or obsolete, thus, the justifications for goal formulation or selection should be monitored in a similar way to plan monitors. The perceptual system should recognize the changes in the state that affect the goal and help the agent to change or retract the goal altogether. The next chapter discusses goal monitors further.

Monitoring the Dynamic Environment: Goal Monitors

The kinds of problems cognitive agents face have increasingly become those in which the agent's goals must be flexible given the dynamic nature of the environments within which they operate. Goals are not simply static predicate representations given as input by some external user. The agent itself is expected to recognize situations in which new goals are to be formulated or current goals changed and abandoned. This is the basic conception of goal reasoning [3, 40].

Cox, Dannenhauer and Kontrakunta [22] discuss a number of operations on goals and distinguish them from operations on plans. Although the purpose of a plan is to establish a state of the world that satisfies a goal or a set of goals, the separation of goal and planning operations provides at least an organizational benefit within a cognitive architecture.

Many planning approaches focus on the capability to generate and execute a sequence of actions that achieve a goal. Some planning approaches replan or adapt plans when the world changes or otherwise is uncooperative. When the goal suddenly becomes true in the current state, some agents will gracefully stop planning or cease executing a plan for a goal that is no longer needed. However few if any address the problem that goals are pursued for some reason. When the reason for the goal (as opposed to the goal itself) ceases to hold, the agent should also abandon the goal or otherwise change its behavior. We propose goal

monitoring as a kind of cognitive process that oversees the continuing benefit of selective goal expressions and when situations warrant decides whether to abandon or change its goals.

In MIDCA, the interpret phase generates goal monitors after adding a goal to the set of pending goals, $\hat{G}$ (see Figure 4.1). These monitors are running asynchronously with other phases in MIDCA to observe the relevant changes to the goal. These monitors inform vision to focus on features/relationships of interest to the goal. If any change happens, the goal monitors reason about it and decide to transform/abandon the goal.

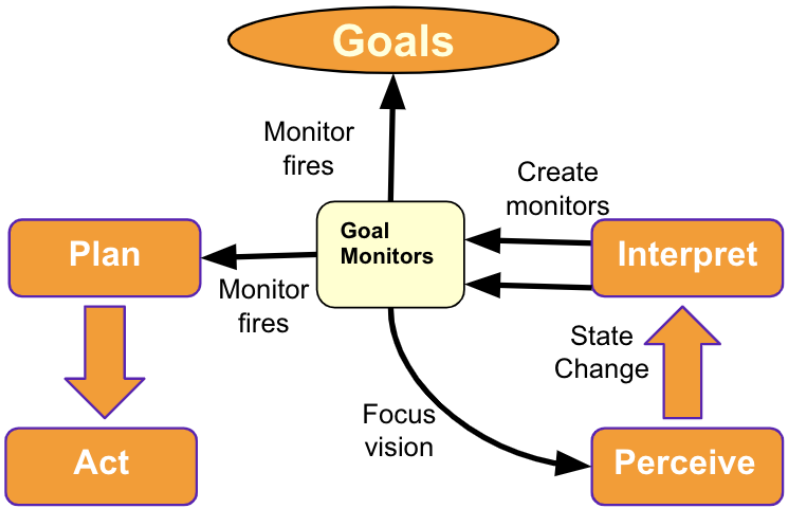

Figure 4.1: A Conceptual representation of perceptual goal monitors. Interpret creates goal monitors after a goal is added to MIDCA. The monitors call perceive. Interpret checks if the perceived information is the same as the expectations. If not, the monitors drop the goal.

We adopt the classical planning formalism [35] where goal is any set of ground literals (i.e., atoms and negated atoms). We say that the agent has achieved its goal g when the current state s satisfies g (i.e., $s \models g$). The agent's goal agenda $\hat{G} = \{g_1, ..., g_c, ..., g_n\}$ contains the current goal g_c and any pending goals it intends to pursue.

In a cognitive system, goals provide focus for the agent's reasoning and represent the desired future state it seeks to achieve. Three types of goal monitors can exist for these

knowledge structures.

1. **Operator-based**. Observing the conditions of rules that generate goals;
2. **Explanation-based**. Observing the causal justifications of the goal;
3. **Direct**. See if the goal is achieved exogenously.

In *operator-based* goal monitors, a set of goal operators as rules generate goals when their conditions are satisfied in the world. *Explanation-based* goal monitors are created from an explanation used to formulate that goal in response to a discrepancy between the agent's expectations and observations. *Direct monitors* check that the goal itself does not exogenously become true at some point in the planning or in plan execution. If they do, then the goal can be dropped from either the set of pending goals or from the current goal expression. In the following sections, we describe Operator-based and Explanation-based goal monitors.

4.1 Operator-based Goal Monitors

We denote a goal operator, o, as the tuple (name(o), precond(o), result(o)). The set of literals precond(o) represents the operator's preconditions. They specify what conditions the current state must satisfy in order for o to be applied. Goal monitors observe these conditions to make sure the goal is still valid in the new state. The term result(o) specifies the goal g. Tac-Air Soar [47] takes this approach. Operators exist for various goal types and data-driven context-sensitive rules spawn them given matching run-time observations.

Algorithm 4 shows high-level details in the MIDCA interpret phase. When a feature being monitored changes and the change is detected, we say that the monitor fires. If a monitor fired, then the goal will be abandoned and removed from pending goals $\hat{G}$ (steps 2-6 in Algorithm 4). The algorithm next checks to see if a new goal is created (step 7). If

a new goal exists and interpret decides to have it monitored, a monitor will be created for g_n's operator (steps 7-9).

Algorithm 4 Goal monitoring in the MIDCA interpret phase. Goal formulation and goal abandonment accompany the monitoring procedure. **Input**: list of pending goals $\hat{G}$, current state s, and list of goal monitors $mnts$. **Output**: list of goal monitors $mnts$ [23]

1: **procedure** $goal\_monitoring(\hat{G}, s, mnts)$
2: **for** g in fired($mnts$) **do**
3: $\hat{G} \leftarrow \hat{G} - g$ ▷ goal abandoned
4: g.is-monitored $\leftarrow \perp$
5: $mnts \leftarrow mnts - (p, g \mid p = monitored\_states(mnts, g)$
6: pending $\leftarrow \hat{G}$ ▷ temp var
7: **if** | pending |+1= | $\hat{G}$ | **then**
8: $mnts \leftarrow$generate-monitors(g_n.created-by,s,$mnts$)
9: return($mnts$)

4.1.1 Monitor Trigger Conditions

Monitor trigger conditions identify the conditions under which the goal should be dropped. Algorithm 5 enumerates the details of creating monitors for the preconditions of a given goal operator o. These preconditions must be monitored because should they become false, the goal is not useful in the current state. This indirectly assumes that goal formulation is performed when the operator preconditions hold in the current state. This procedure is similar to Algorithm 2 for creating plan monitors. The difference is that line 5 and 6 in Algorithm 2 are not needed for goal monitor creation, because if a condition is not satisfied in the state the goal will not be created in the first place.

4.1.2 Monitor Response

Algorithm 6 checks for monitors that trigger. It calls perceive with the goal precondition p as an input argument to get the current state s (perceive only updates the features relevant

Algorithm 5 Goal-monitor generation. The algorithm assumes that the monitor uses the operator style. **Input**: goal operator o, world state s, and list of monitors $mnts$. **Output**: list of monitors $mnts$ [23].

1: **procedure** $generate\_monitors(o, s, mnts)$
2: $mnts \leftarrow \langle\rangle$
3: **for** p in $precond(o)$ **do**
4: $mnts \leftarrow (p, g) \cup mnts$
5: return $mnts$

to p) (step 4). It then takes a list of monitors and checks if the conditions are still satisfied in s. If not, it will assemble a list of goals to drop from the list of the agent's pending goals.

Algorithm 6 Check for fired monitors. **Input**: list of monitors $mnts$. **Output**: list of goals to drop $goals\_to\_drop$ [23].

1: **procedure** $G\_fired(mnts)$
2: goals-to-drops $\leftarrow \emptyset$
3: **for** (p, g) in $mnts$ **do**
4: $s \leftarrow$ perceive(Ψ, p)
5: **if** $s \not\models p$ **then**
6: $goals\_to\_drop \leftarrow g \cup goals\_to\_drop$
7: return $goals\_to\_drop$

4.1.3 A Short Goal Monitoring Example

Table 4.1.3 illustrates an example goal operator for a logistics delivery task. If MIDCA receives an order to deliver package p_1 to location l_{11} and p_1 is available in one of the warehouses (e.g., w_2), the *beta* function uses the goal operator to generate the delivery goal g_7= *delivered*(p_1, l_{11}). If MIDCA decides to monitor this goal, monitors are created to observe the conditions *obj-at*(p_1, w_2) and *order*(p_1, l_{11}). Now if at a later time, p_1 is stolen or missing from the warehouse or the order is canceled, then the monitor will abandon g_7, removing it from $\hat{G}$.

Attribute	Representation
Goal operator	$o(?p, ?w, ?l)$
Preconditions	$\{obj\text{-}at(?p, ?w), order(?p, ?l), delivered(?p, ?l) \notin \hat{G}\}$
Result	$g = delivered(?p, ?l)$
Monitor conditions	$\{obj\text{-}at(?p, ?w), order(?p, ?l)\}$

Table 4.1: Example goal operator for delivering an ordered package.

4.2 Explanation-based Goal Monitors

Explanation-based goal monitors focuse on *goal-driven autonomous (GDA)* agents. Goal-driven autonomy involves recognizing unexpected or possibly new problems, explaining the causal factors underlying the problems and generating goals to remove the cause of the problems in order to achieve the given task. The GDA agent itself is expected to identify situations in which new goals are to be formulated or current goals changed or abandoned [22, 48]. Identification of these situations is where the explanation-based goal monitors are needed.

In partially observable domains, the GDA agent might consider multiple hypotheses. If there are two or more possible explanations, each could be equally likely initially. But if relevant new information is observed, the system should consider updating the explanations. A contribution of this thesis is handling multiple explanations in situations where more than one hypothesis may be true. The goal monitors are created for all possible hypotheses, and when the agent observes new information, one or more of the goal monitors may fire which in turn will lead to a possible change in the agent's belief and its goals.

Explanation-based goal monitors link the causal information of the explanations to both (1) the newly generated goals and (2) monitoring capabilities for identifying if the goals remain relevant. In the following sections, we describe the process of creating goal monitors in the MIDCA cognitive architecture and explain how they lead to goal change among competing goals when anomalies arise.

We will now describe the execution monitoring framework for an agent equipped with explanation based goal monitors. First, we describe how the explanation module generates

the necessary information for goal monitors. Next, we describe how the monitors use this information to decide if the current goal is still valid in a dynamic world, and which goal the agent should pursue when there are multiple possible goals. Finally, we describe how the framework will respond to changes in the conditions that are being monitored.

4.2.1 Explanation

In this work, we define the term explanation as a set of assumptions about the unobserved facts. MIDCA employs DiscoverHistory to generate these explanations using causal inference over the environment model. DiscoverHistory reasons about the causes of inconsistencies between observations and expectations that arise during plan execution and generate abductive explanations.

An occurrence o is an occurrence of any observation obs, action α, or event e. An execution history is a finite sequence of observations and actions $\langle obs_0, \alpha_1, obs_1, \alpha_2, ..., \alpha_k, obs_{k+1}\rangle$. An inconsistency is a tuple (p, o, o'), where p is a predicate in the new observation o' ($p \in o'$) but p is not an effect of the previous occurrence o.

At each cycle of MIDCA, the execution history $\langle obs_0, \alpha_1, ..., \alpha_k, obs_{k+1}\rangle$ is passed to DiscoverHistory to detect discrepancies. When a new observation is inconsistent with the agent's expectation ($\neg p \in$ effect(α_k) and $p \in obs_{k+1}$), DiscoverHistory tries to resolve the inconsistency. DiscoverHistory deduces the possible worlds that result from different sets of assumptions.

One way to resolve the inconsistency (p, o, o') is to show that some occurrence changed the value of a literal in between the preceding occurrence o and the following occurrence o'. This occurrence must be an event e, such that effects$(e) \models p$. Another way to resolve the inconsistency is hypothesizing an initial value. In this case, a different initial occurrence o may be hypothesized. We use DiscoverHistory to generate these hypotheses. Example 7

shows a scenario where DiscoverHistory resolves an inconsistency.

Example 7. Detecting an Anomaly In the Minecraft domain, suppose that Steve moves to a location m_i and observes his health is lower than expected. Steve finds this anomalous because he is expecting good health (a health-value $\geq$ 30), but instead observes a lower health value. Figure 4.2 illustrates the inconsistency corresponding to this situation. There are three occurrences including two observations (o_i and o_{i+2}), and a move action (o_{i+1}). The move action should not affect the agent's health, and the expected value for health is 30, but the subsequent observation contradicts this. One way to resolve the inconsistency is to show that some event changed the value of health.

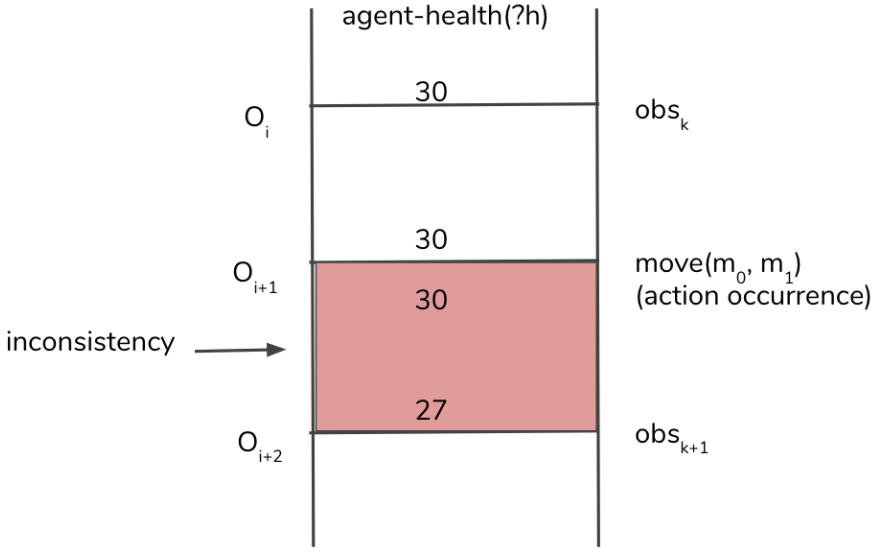

Figure 4.2: Relevant action and event descriptions are given on the right. The expectation and observed value for health are given in the timeline; for example, the value 30 at the top indicates that the health value is 30.

The event *attack-skeleton* has an effect that causes Steve's health to go down by three. This event could be added to the explanation to resolve the inconsistency. In order for this to work, a new occurrence must be added between o_{i+1} and o_{i+2} (see Figure 4.3). The pre-

conditions for this event are not valid in the current state, so the new inconsistency requires DiscoverHistory to hypothesize an initial state assumption. Therefore, the discrepancy can be resolved by adding the initial state assumption *skeleton-at(adj-m_1)* (see Figure 4.3).

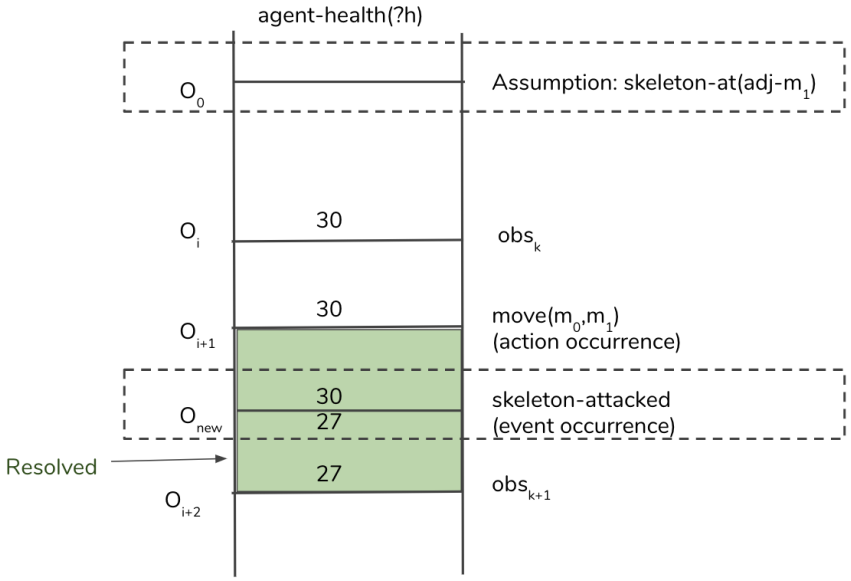

Figure 4.3: Example of resolving inconsistencies by hypothesizing an initial value and adding an occurrence

Explanation Generation

Each possible world is described by one explanation which includes new assumptions about the world that resolve the discrepancy. Let's assume that there are n possible explanations, each has m assumed initial values $h = [p'_1, ..., p'_m]$ and one possible event $e_i, i \in [1..n]$. Assumptions across all explanations are added to the MIDCA's set of hypotheses $hyps = [h_1 ... h_n]$.

In our scenario described in Example 7, the appearance of a skeleton leads to a skeleton attack event e_1 that includes effects that match the discrepancies in the observed state (effect(e_1) $\models p$, where p is low health). DiscoverHistory produces an explanation that includes the unobservable facts that would cause the event to occur. MIDCA adopts these

facts as hypotheses. Example 8 shows the output of DiscoverHistory for the scenario in Example 7.

Example 8. Explanations generated by DiscoverHistory

```
Explanation 1:
(ASSUME-INITIAL-VALUE
  (THING-AT SKELETON ADJ-M) T TIME 1)
(SKELETON-ATTACKED M1-1 ADJ-M 27 30
                   TIME 8)

Explanation 2:
(ASSUME-INITIAL-VALUE
  (IS-TRAP ADJ-M) T TIME 1)
(FALL-IN-TRAP M1-1 ADJ-M 27 30 TIME 8)
```

These two hypothetical sets of facts are competing alternatives. The first explanation states that there is a skeleton in one of the tiles adjacent to the agent, and an event e = `SKELETON-ATTACKED` occurred. The second explanation states that there is a trap in one of the tiles adjacent to the agent. Note that `(THING-AT SKELETON ADJ-M)` and `(IS-TRAP ADJ-M)` are unobservable facts. MIDCA adds these assumed predicates to the set of hypotheses. The set of hypotheses includes the predicates that are not believed to be true by the agent yet. Although the agent assumes that these predicates could have been true in the state.

4.2.2 Monitor Trigger Conditions

An abstract goal g_a is generated in the interpret phase in MIDCA in response to a discrepancy. After explaining the discrepancy, the agent formulates goals for each explanation (if any) and adds these goals to the set of g_a's subgoals. Algorithm 7 shows how g_a is decomposed into a set of subgoals $\langle g_1, ..., g_n \rangle$ using the causal structure of each explanation. It takes as input the abstract goal, g_a and the set of hypotheses for a discrepancy, $hyps$. First, the set of hypotheses $hyps$ is sorted based on some metrics like their possibility, danger,

etc (line 2). Then, for each hypothesis a goal is formulated and is added to the set of g_a's subgoals (lines 4-5). Next, if interpret decides to have a goal monitored a goal monitor for each assumed predicate in a hypothesis is created (lines 6-8). The first subgoal is selected as the current goal (line 9). Example 9 shows the set of hypotheses, goals and goal monitors for example 7.

Example 9. Hypotheses, Goals, and Monitors

h_1 = THING-AT(SKELETON,ADJ-M)

h_2 = IS-TRAP(ADJ-M)

g_1 = ¬THING-AT(SKELETON,ADJ-M)

g_2 = ¬IS-TRAP(ADJ-M)

g_a = $\geq$ current-health 30

m_1= $\langle$ THING-AT(SKELETON,ADJ-M), g_a, $g_1\rangle$

m_2= $\langle$IS-TRAP(ADJ-M), g_a, $g_2\rangle$

When a goal g_i is generated in response to event e_i, the interpret phase of MIDCA formulates an explanation-based goal monitor GM for each predicate in the hypothesis h=$[p'_1, ..., p'_m]$. GM is a tuple (p', g_a, g_i) that consists of the predicate p' from a hypothesis related to event e_i, the abstract goal g_a, and the formulated goal g_i. The hypotheses provide the environmental conditions that must persist for the goal g_i to remain valid. Goal monitors observe these conditions and provide the response if they change. If these conditions change in the state, the goal monitor will fire, and the GDA process for goal management will know to reconsider pursuing the current goal.

Algorithm 7 Elaborate the abstract goal and create explanation-based goal monitors. **Input:** Abstract goal g_a, set of hypotheses $hyps$.

```
1: procedure ELABORATE-GOAL(g_a, hyps)
2:     sort hypotheses based on possibility
3:     for h in hyps do
4:         g_i ← formulate goal
5:         g_a.subgoals ← g_a.subgoals ∪ g_i
6:         for p' in h do
7:             mnts ← mnts ∪ (p', g_a, g_i)
8:     g_c ← g_a.subgoals[1]
```

4.2.3 Monitor Response

Algorithm 8 shows the firing condition for the explanation-based goal monitors. It takes as input the abstract goal, g_a and the set of goal monitors $mnts$. Goal monitors tie the predicates from hypotheses to goals, and then when a monitor fires the agent consider switching the goal. The first step in monitoring is to check if the hypothesized predicates are in the new observed state.

If a monitored predicate p' is observed, MIDCA adds that to the set of beliefs, s_b. If p' is an effect of event e and goal g, then the current goal should change to g since e is what really happened in the world (lines 3-4).

If a monitored predicate p' is believed to be false ($\neg p \in s_b$) then the corresponding goal should be abandoned (if it is current goal) and removed from the set of subgoals. Then, the algorithm changes the current goal to be another possible subgoal of g_a (lines 5-8). If no subgoals exist in g_a, it will abandon g_a.

Algorithm 8 Check for fired Goal monitors. **Input:** Abstract goal g_a, set of monitors $mnts$.

```
1: procedure FIRE-MONITOR(g_a, mnts)
2:     for (p', g_a, g) in mnts do
3:         if p' ∈ s_b then
4:             g_c ← g
5:         if ¬p' ∈ s_b then
6:             g_a.subgoals ← g_a.subgoals − g
7:             if g_c = g then
8:                 g_c ← g_a.subgoals[1]
```

Goal monitors can handle a situation where multiple hypotheses exist, and by monitoring each of them, the agent can switch goals if these hypotheses are believed to be true or false. The reasons why the agent pursues a goal are extracted from these hypotheses automatically. In the scenario in example 7, the goal is to have health recovered. The reason why the health was low in the first place could be the existence of a trap or a skeleton. The agent assumes that a trap or a skeleton is present so he picks one based on some metrics (e.g., danger) and pursues that. If a trap is present, then he does not need to pursue the skeleton goal. If neither a trap or skeleton are present, then the agent does not need to achieve this goal. Possibly a one time event occurred like an explosion of a creeper.

We present empirical results within the MIDCA cognitive architecture using the monitors to focus perception, adapt plans, and change goals in the next two chapters.

4.3 Discussion

An autonomous agent not only needs to generate plans to achieve different goals but also detect problems, formulate new goals, monitor the plans and goals, change the goal and replan when its beliefs change in order to operate in complex environments. In this chapter, we introduced an execution monitoring framework for an agent equipped with goal monitors. Our system supports monitoring during execution, and it detects unexpected changes in the agent's goals. We discussed three different types of goal monitors and explained how

an autonomous agent is more robust using these monitors in dynamic environments.

These monitors anticipate and describe the circumstances under which the agent needs to reconsider its goals and describe how the agent should respond when these situations arise. In addition, Explanation-based Goal monitors can help an agent to respond appropriately in the presence of multiple competing hypotheses, and by monitoring each of them the agent can switch goals if they are believed to be true or false.

This work focuses on high-level cognition, represents goals as structured knowledge, and is inspired by human cognition. This research represents the functional roles goal operations contribute to successful high-level reasoning and subsequent robust behavior for cognitive agents in difficult environments.

The Evaluation of Plan Monitors

In this and the next chapter, we investigate claim 1 and claim 2 presented in chapter 1. Here, we provide experimental evidence for these claims through three experiments in two domains (i.e., blocksworld and logistics).

- *There are measurable advantages in integrating interpretation, planning, and perception in a cognitive architecture. We show that these cognitive processes are dependent.*
- *Plan monitors allow the agent to anticipate failures with the plan in a dynamic world and guarantee that a valid plan is produced when plan generation terminates. This method improves the planning performance of an agent in complex domains.*

5.1 Simulated Domains

The blocksworld and logistics domains were used in the experiments in this chapter. We use MIDCA's standard world simulator which simulates actions specified using predicate logic. The types of actions that can be performed are specified prior to startup in a domain file (see Appendix A and C). Actions produced during the Act phase will be simulated, as well as actions performed by other agents and natural events.

5.1.1 Blocksworld

Our version of blocksworld includes both triangular and rectangular blocks, which compose the materials for simplified housing construction. The initial goals for problems in this domain are to build houses consisting of towers of blocks with a roof (triangle) on each.

We added the possibility that blocks could catch fire, and before picking up any block, the fire should first be extinguished. In order for an extinguisher to be used, it must first be taken out of a box. The box itself is represented as a block. If the box is not clear, the planner generates a sequence of actions(i.e., a subplan) to make the box clear. Furthermore, we implemented three additional actions allowing MIDCA to deal with these refinements. The three new types of actions are as follows:

extinguish *(*C, ext*)* : extinguish block C using the extinguisher ext

preconditions: on-fire(C), holding(ext)

get-extinguisher *(*ext*, *B*)* : take out the extinguisher ext from B

preconditions: *clear(*B*), in-box(*ext*, *B*)*

make-box-clear *(*B*)* : unstack all blocks on top of B

preconditions: $\neg$ *clear(*B*)*

5.1.2 Logistics

The logistics domain [72] represents a simplified shipping problem involving planes, trucks, cities, and airports. Goals generally specify the destination of packages, and plans describe the actions needed to get a package to a location.

In our modified logistics domain, we added a new predicate *adj* which specifies which cities are adjacent (each city has at most one adjacent city). We also added two types of planes: regional and transcontinental. The regional planes can only fly to an adjacent city,

but large transcontinental planes can fly without limit. If there is a large plane in an airport, the planner will choose that over a regional plane.

The new operators added to the domain are as follows:

fly-airplane(a_1, c_1, c_3) : fly to an adjacent city with a regional airplane

preconditions: *airplane-at* (a_1, c_1), $\neg$ *largeairplane-at* (a_2, c_1), *adj*(c_1, c_3)

fly-largeairplane(a_1, c_1, c_3) : fly with a large airplane

preconditions: *largeairplane-at* $(a_1, c_1))$

For example, consider there are three cities (*adj* (c_1, c_2), *adj* (c_2, c_3)), and a package, p_1, at c_1 (*obj-at*(p_1, c_1)). The goal is *obj-at*(p_1, c_3). If there are only regional planes available at c_1, two flights are needed from c_1 to c_3 (see Figure 5.1). If there is a large plane available at c_1, then only one flight is needed from c_1 to c_3 to transfer the package to the destination (see Figure 5.2).

Figure 5.1: Logistics example to move a package from c_1 to c_3 with regional planes

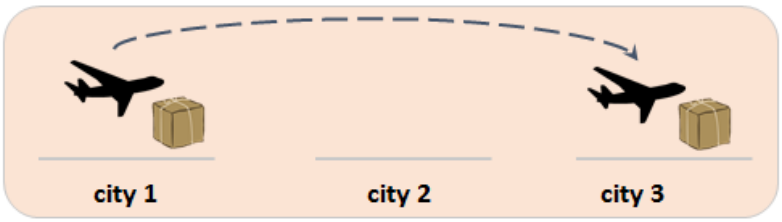

Figure 5.2: Logistics example to move a package from c_1 to c_3 with a transcontinental plane

5.2 Evaluation: Perception Helps Planning

MIDCA_1.4 is the latest implementation of the MIDCA architecture which was described in chapter 2. In this section, we describe our experiments with MIDCA_1.4 on a modified blocksworld domain [32, 64, 77] and a modified logistic domain [72]. We then report results from three experimental scenarios regarding the performance of plan monitors.

We claim that, especially in dynamic environments, rationale-based perceptual plan monitors are key when integrating two major cognitive mechanisms: (1) planning for action in the world and (2) interpretation or making sense of the world. To evaluate the role these monitors play in the mutual relationship between the two, we can examine how perceptual plan monitors help planning and interpretation, or conversely, we can look at how planning and interpretation help perception.

5.2.1 Blocksworld Experiments: Scenario one

In this experiment, we changed the world state in the middle of planning to satisfy a previously unsatisfied precondition. This invalidates the current plan that included steps to satisfy the condition, and therefore, the change requires the unnecessary steps to be removed. The monitor notifies the planner about the change, and the planner will transform the plan to successfully achieve the goal. Note that this plan transformation is an instance of the second type mentioned in Section 3.1 (i.e., shortening the plan by removing steps).

In each planning problem, we set the initial state to be one with a block R on fire, a separate tower with B as its bottom-most block, and a fire extinguisher, ext, inside B. The goal is *on(R,G)*. Figure 5.3 (a) shows an example for this problem (the height of the tower here is 3). Block R is on fire, *onfire(R)* is true, and in order to pickup R, the fire needs to be extinguished first. Since the height of the tower is 3, the planner has to unstack and put down 2 blocks, X_1 and X_2, in order to obtain the fire extinguisher from block B and use it on block R (Figure 5.3(b) and (c)). If the fire goes out during the planning process, the

monitor watching the precondition *onfire(R)* fires. Then the planner cuts parts of the plan related to extinguishing the fire and simply generates the plan *pickup(R)*, *stack(R,G)*.

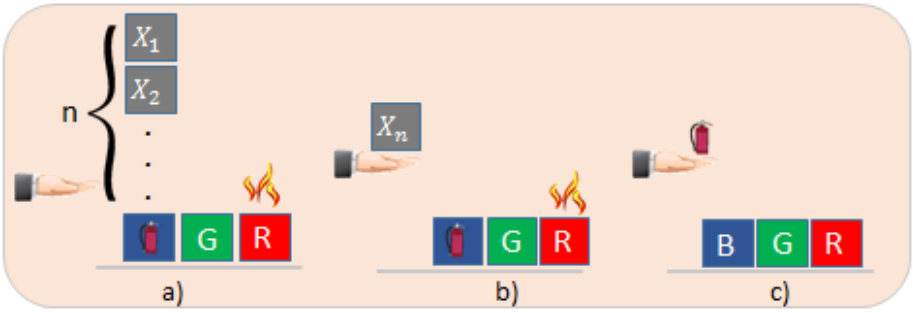

Figure 5.3: A Blocksworld example with the goal of having block R on G. a) Initial state has block R on fire, G on the table, and the extinguisher under a tower of blocks. b) The plan is to unstack the tower to access the fire extinguisher, c) and then put out the fire so R can be put on G.

Here, the purpose of monitoring is to observe such a change as the fire going out, and suggest a cut in the plan. By varying the height of the tower, we can vary the complexity and length of the solution. In this experiment, we varied the height of the tower, n, from 10 to 50 in increments of 10. During planning, the monitor is observing the state of *onfire(R)* fires and suggests a plan refinement. We vary the time at which this monitor fires during the planning process, namely after 10, 70, 110, and 170 planning steps.

Experimental Results for Scenario One

Figure 5.4 shows the results of the experiment and plots the planning steps as a function of n. The dotted lines show that planning is over before the change happens. When the environment does not change, the number of planning steps increases with n. However, with the rationale-based monitors, the planner can react to the state changes and find a solution faster. As expected, when the changes occur later, the savings benefit of the planner is reduced because it has already performed significant planning. When the delay is infinite, it has to unstack all the blocks to get the extinguisher. When the delay equals 10, the fire

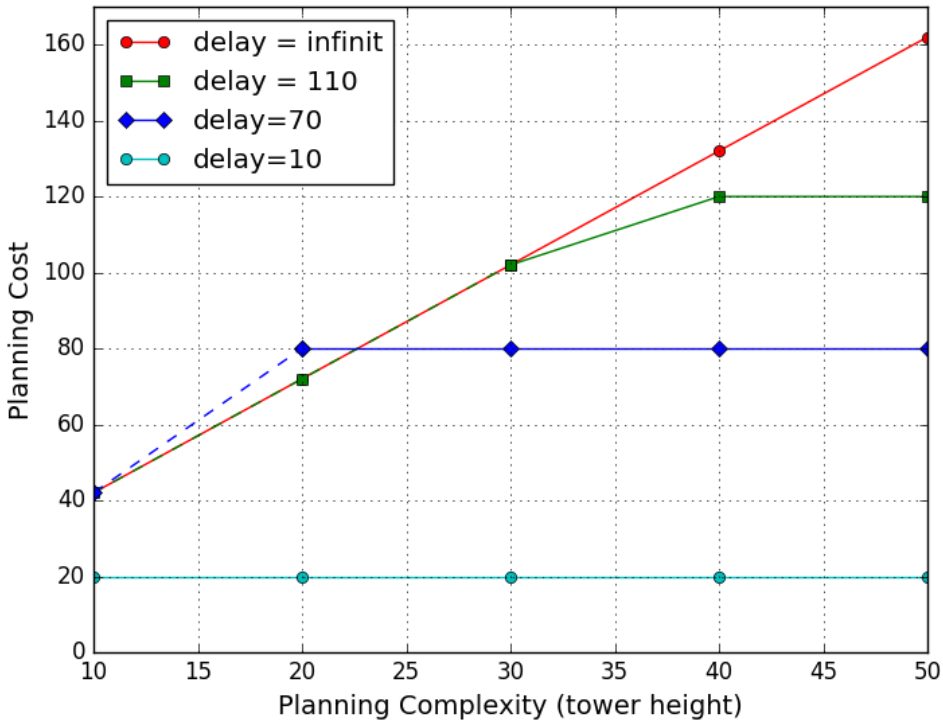

Figure 5.4: Planning performance in blocksworld using perceptual plan monitors with a single type of plan transformation (i.e., step removal). The curves refer to different delays of the state change during the planning process. The dotted lines show that planning is over before the change happens.

goes out in the very beginning and the number of planning steps will be 20 for all towers regardless of their height.

5.2.2 Blocksworld Experiments: Scenario two

In this scenario, the goal is *on*(R, G). The agent should unstack all the blocks in the tower with height n to pickup block R and then stack R on G (see Figure 5.5 (a). In the middle of planning, another agent removes m blocks from the tower and puts them on top of block G (see Figure 5.5(b)). The plan monitors detect the change since the preconditions of earlier steps in the plan are violated. The planner modifies the plan with cutting the steps "unstack m blocks from the red block R" and adding steps "unstack m blocks from the green block

G". Note that this plan transformation is an instance of the first and second type mentioned in Section 3.1 (i.e., extending the plan with additional steps and shortening the plan by removing steps).

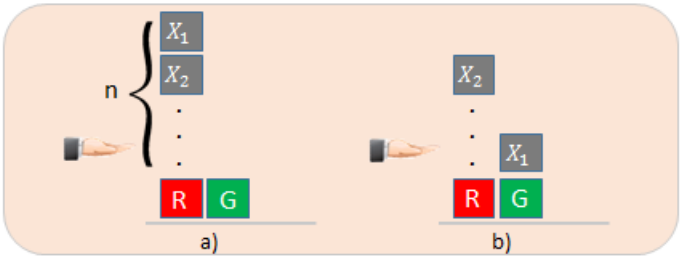

Figure 5.5: A blocksworld problem to stack block R on G. The first panel shows the initial state. The second panel shows the state when another agent moves a block from the first tower to the second block G.

Experimental Results for Scenario Two

Figure 5.6 shows the results of this experiment. The dotted lines show that planning is over before the change happens. For example, when the delay is 200 and the height of tower is 20, the number of actions to construct the plan is 180. The chart shows as the delay happens later, the amount of saving in planning is reduced. This proves our claim that using plan monitors save planning cost in a dynamic world.

5.2.3 Logistics Experiments: Scenario three

We set the initial state to be one with a set of n adjacent cities, and regional planes in each airport of the cities. The task is transferring the package from c_1 to c_n, where n flights are needed to accomplish this task. In our experiment, we make a large plane available at c_3. When the large plane becomes available, the monitor detects that change and suggests a shorter plan using the large airplane. The large airplane can fly straight from c_3 to c_n.

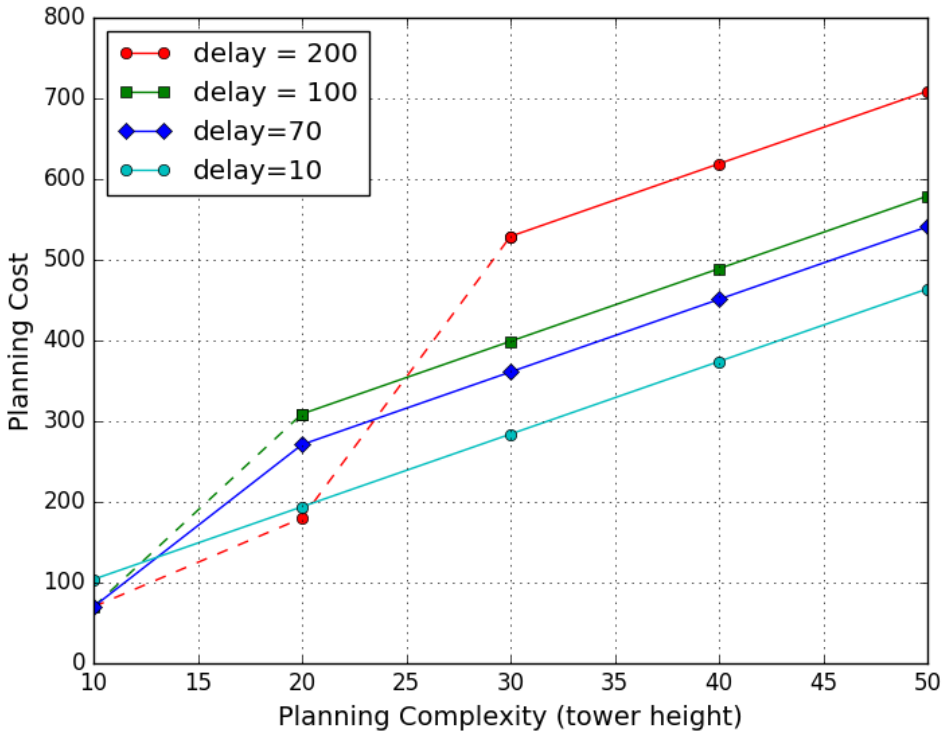

Figure 5.6: Planning performance in blocksworld using multiple perceptual plan monitors; some remove steps and others add steps. The curves refer to different delays of the state change during the planning process. The dotted lines show that planning is over before the change happens.

We varied the number of cities between the package's location and its destination, n, from 10 to 50 in increments of 10. During planning when a large plane becomes available, the monitor fires and suggests a plan refinement. We vary the time at which this monitor fires during the planning process, namely after 10, 70, 110, 170 planning steps.

Experimental Results for Scenario Three

Figure 5.7 shows the results of the experiment and plots the planning steps as a function of n. When the delay is infinite (there is no large plane), the plan needs n flights. When the delay equals 10, the large plane will be chosen in the very beginning and only three flights are needed (two flights from c_1 to c_3 and one flight from c_3 to c_n).

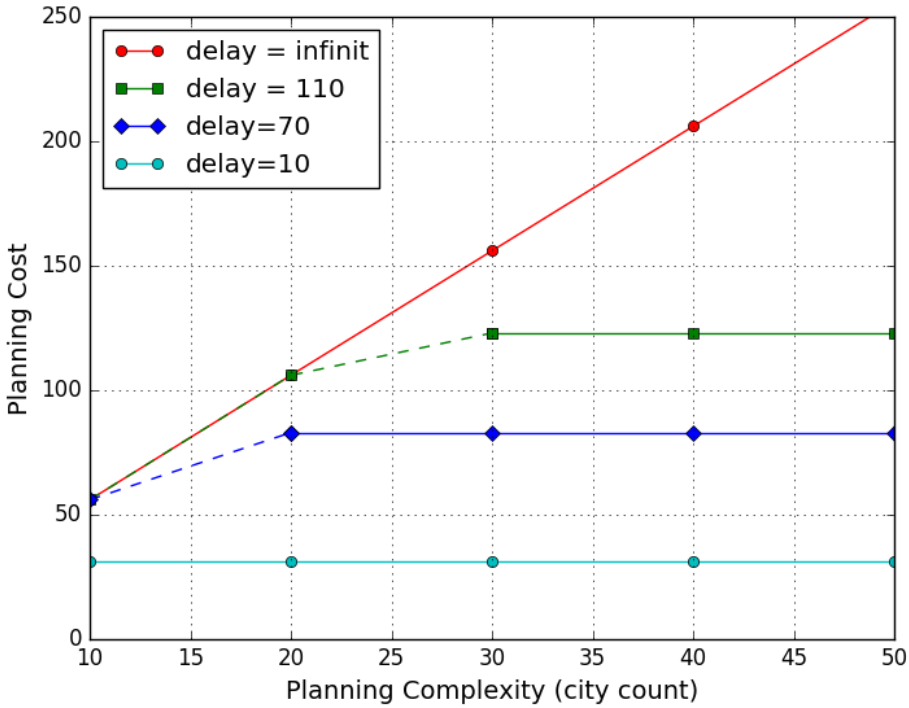

Figure 5.7: Planning performance in logistics domain using perceptual plan monitors with a single type of plan transformation (i.e., step removal). The curves refer to different delays of the state change during the planning process. The dotted lines show that planning is over before the change happens.

5.3 Evaluation: Planning Helps Perception

When perception is acting independently, it finds all the objects and builds all the relationships between them. We show that perception can be more efficient and save many steps when it is biased to the agent's goal and plan. Figure 5.8 shows an example where there are n blocks on the table. The agent's goal is $on(A, B)$, and the plan to achieve the goal is $\langle$ *pickup(A), stack(A, B)* $\rangle$. Based on the generated plan, the agent only needs to focus on blocks A and B and their relationships with other blocks. Any change that happens in the other n blocks, this will not effect the agent's success in achieving its goal. Plan monitors help the agent to focus only on the related features in the world.

Table 5.1 compares the number of predicates generated by the perception module in

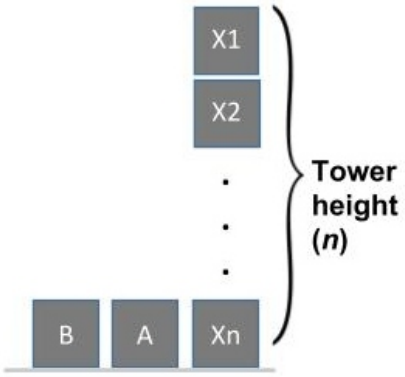

Figure 5.8: Planning guides vision to focus on what the agent is doing. The agent's goal is *on(*A, B*)*.

an agent with plan monitors and a baseline agent without plan monitors. The baseline agent looks for all the object in the scene and creates all the predicates, while our agent only focuses on the two objects related to the goal. As shown in Table 5.1 , the perception module only needs to create and observe three predicates. This is a general benefit of using plan monitors. Although, there might be some situations that focusing vision on certain features of the world leads to missing some information about the environment. For example, if a block that is not relevant to the agent's current goal catches on fire, the agent would miss that, although it has some effect on its future goal.

	Cost	**Predicates**
Using Plan Monitors	3	clear(A); clear(B);on(A, table)
No Plan Monitors	$n + 5$	clear(X_1); on(X_1, X_2), . . . , on(X_{n-1}, X_n); clear(A); clear(B); on(A, table); on(B, table); on(X_n, table)

Table 5.1: Planning guides vision to be biased to the agent's goal. The agent's goal is *on(*A, B*)*.

The Evaluation of Goal Monitors

In this chapter, we investigate claim 1 and claim 3 presented in chapter 1. Here, we provide experimental evidence for these claims through two experiments in two domains (i.e., minecraft and logistics).

- *There are measurable advantages in integrating interpretation, planning, action and perception in a cognitive architecture. We show that these cognitive processes are dependent.*
- *Goal monitors improve the goal achievement performance of a cognitive agent in a partially-observable, dynamic environment. This approach provides the basis to change the goal when the goal is no longer useful in the world.*

6.1 Simulated Domains

The following domains were used in the experiments in this chapter (see Appendix D and E). The logistic domain is fully observable and dynamic. The Minecraft domain is partially observable and dynamic.

6.1.1 Logistics

The version of the logistics domain [72] we use includes trucks or airplanes delivering packages from different warehouses to various destinations. The agent is tasked to deliver packages for different orders. For example, transporting the package p_1 by truck to location l and then unloading it achieves the goal g_c= *delivered*(p_1,l). We assume that the agent has full observability and has access to the list of packages in the warehouses.

6.1.2 Minecraft

In the Minecraft game, the character named Steve explores an infinite 3D virtual world while gathering resources and surviving dangers. We used a simulated version of Minecraft which is 2D and finite [11] [1]. Different factors can damage Steve's health like falling in lava, getting shot by a skeleton archer, triggering an arrow trap, and low hunger level. In this evaluation, we only consider two possible events: getting shot by a skeleton archer or triggering an arrow trap. Both events cause three damage points to the agent's health.

6.2 The Evaluation of Operator-style Goal Monitors in Simulation

We claim that using operator-based goal monitors in a cognitive architecture like MIDCA increases the number of goals the agent can achieve. To evaluate this hypothesis, we conducted tests with MIDCA on a simulated logistics domain. We use a simulator to model the world state and agent actions that change the state.

[1] The files that provide PDDL representations are found at http://groups.csail.mit.edu/rbg/code/planning/

6.2.1 Logistics Experiments

When an order for a package exists and the package is present in one of the warehouses, Interpret generates a delivery goal for that package. The MIDCA Intend phase selects one warehouse and commits to achieving all goals for the packages in that warehouse. The JSHOP planner [62] that implements the MIDCA plan phase then generates a plan for these packages. If one package is stolen from a warehouse w_i, the planner fails to generate a plan for all delivery goals in w_i. However, with goal monitors, the agent will know that a package is missing, and before Plan starts planning for that warehouse, it will drop the goal for the missing package. Planning will now succeed for the current goals. Notice that lost packages are distributed evenly across warehouses, and we assume that packages are stolen from the warehouse that it is not planning for currently.

We ran two experiments: In the first, we varied the number of warehouses, and in the second, we varied the number of lost packages. Every goal achieved (each package delivered) by MIDCA has a score of one point. In each scenario of the first experiment, we set the initial state to be the one with n warehouses, five packages in each warehouse, and one order exists for each package. Interpret generates a delivery goal for each package. During runtime, six packages from different warehouses were lost. We varied the number of warehouses from five to twenty in increments of five. Each warehouse has five packages. In each scenario of the second experiment, the initial state is set to be the one with twenty warehouses with five packages each. During runtime, n packages were lost. We varied the number of lost packages from one to twenty.

Experimental Results

Figures 6.1 and 6.2 summarize the results of MIDCA with and without goal monitors for two scenarios. The y-axis is the goal score that the agent was able to achieve for delivering packages.

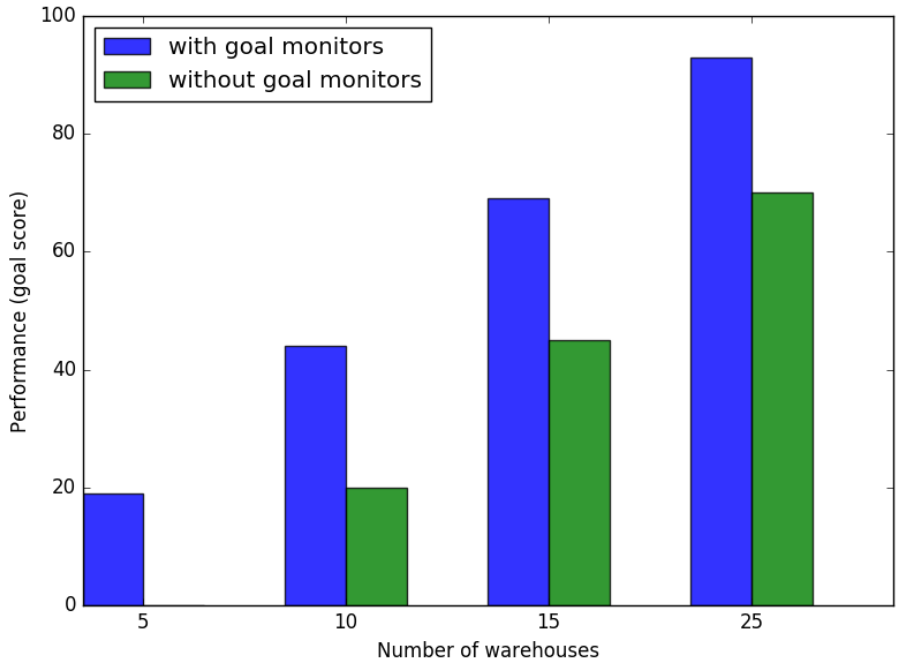

Figure 6.1: Logistics domain performance with goal monitors and without goal monitors. Six packages from different warehouses were lost. Each warehouse has five packages. In case the number of warehouse is five, the number of goal achieved for the agent without goal monitors is zero.

We plot the score as a function of the number of warehouses in Figure 6.1 and in Figure 6.2 as a function of the number of lost packages. The results show that the performance of MIDCA with goal monitors is better than MIDCA with static goals (e.g., no goal monitors), because goal monitors allow the agent to drop its goals when they are not achievable. In Figure 6.1, when the number of warehouses is five, MIDCA without goal monitors is not able to achieve any goal (one package is lost from each warehouse causing all plans to fail).

Figure 6.2 shows the result of the second scenario with twenty warehouses. When no package is lost, both approaches show equivalent performance. But when more packages are stolen, MIDCA with goal monitors is able to achieve a higher score by dropping delivery goals of lost packages. The score of MIDCA with static goals converges to zero as more packages are lost. The results support our claim that the goal monitors technique improves the performance of a cognitive architecture in a simulated logistics domain.

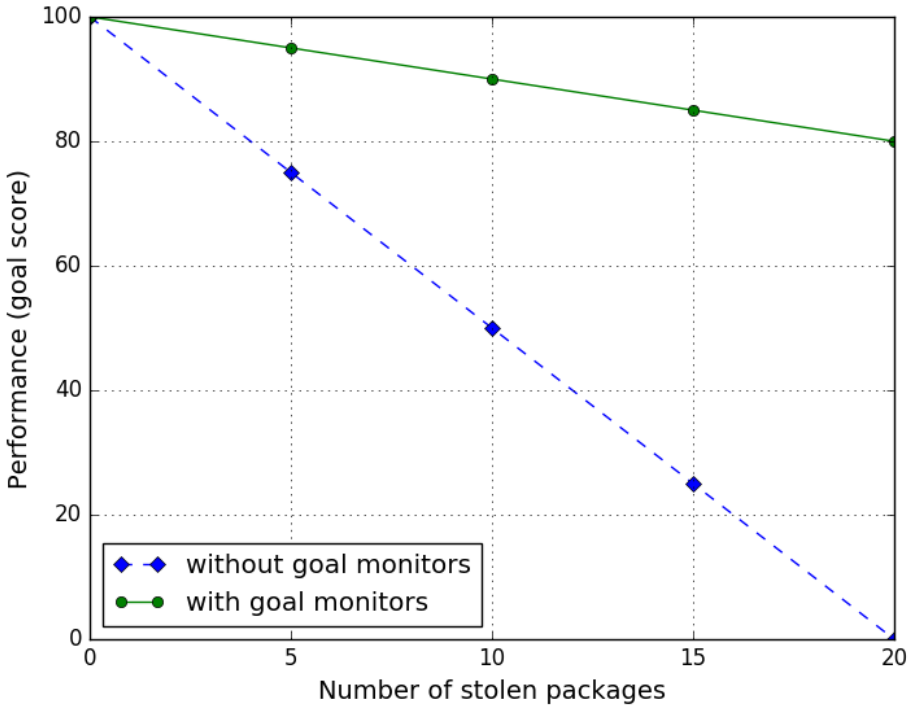

Figure 6.2: Logistics domain performance in MIDCA for twenty warehouses with five packages in each.

6.3 The Evaluation of Explanation-based Goal Monitors in Simulation

We claim that in partially observable domains with unexpected events an agent with goal monitors will outperform an agent without goal monitors. To evaluate this hypothesis, we conducted tests with MIDCA on a partially observable Minecraft-based domain. The MIDCA agents operate in planning domains using a PDDL domain definition [33] representing actions, events, and predicates. For planning, we use the MetricFF planner [41].

In the Minecraft domain, various unobservable events occur that our agent does not have sufficient knowledge to predict. To evaluate the impact of goal monitors, we compare our agent *GMAgent*, who is equipped with explanation-based goal monitors against two baselines: a GDA agent with explanation capabilities but no goal monitors which we

call *EXAgent*, and an off-line planning agent named *PLAgent*. In this scenario, GMAgent should outperform PLAgent due to the environments' partial observability. EXAgent generates explanations when an anomaly happens and formulates goals in response to the anomalies, but it does not monitor the current goal or the set of pending goals. We hypothesize that GMAgent will outperform the EXAgent in scenarios where unobserved facts affect the state and multiple hypotheses exist, due to goal monitoring, and it will outperform PLAgent when events occur outside of the agent's mission due to goal reasoning. We measure performance as progress towards achieving the initial goal which we describe next.

6.3.1 Minecraft Experiments

The agent is tasked with the initial goal to obtain seven pieces of wood. Wood is obtained by harvesting trees via a chopping action which requires an axe. The locations of trees are known by the agent a priori. We now walk through a scenario in which Steve should switch to a pending goal.

While Steve is en route to a known tree's location, he observes his health has suddenly decreased and his life is in danger. Steve's health begins at a value of thirty. When Steve's health value reaches zero, the agent dies and no more goals can be accomplished. Performance is calculated based on how many trees the agent can harvest. We gave the agent three lives; whenever the agent dies, he looses all his items and he starts from the beginning tile. However, the agent retains credit for trees harvested before death.

The PDDL planning model for this experiment consists of fifteen actions, and two events. We randomly generated fifty problems for each difficulty level. Difficulty levels differ only in the number of random traps placed. More skeletons could make the scenario more difficult, but since two agents, EXAgent and GMAgent, behave the same, we only make the scenario more difficult by varying the number of traps. In each problem, there are seven trees, and five skeletons. Each problem takes place in a 10×10 grid. The number of

traps (or difficulty level) was varied from zero to six. In this experiment, if the agent goes to an adjacent tile with a skeleton or trap, he gets shot by an arrow. If there is a skeleton, the agent needs to shoot the archer skeleton using a bow. If the agent triggers a trap, then he needs to disarm the trap with an axe. Note that skeletons and traps are unobservable and the agent can see them only after performing sensing actions and facing the direction of the skeleton/trap.

Experimental Results

In each difficulty level, we performed 50 trials which were held constant across the three agents. Figure 6.3 shows the average performance of GMAgent, EXAgent and PLAgent. When there is no trap, GMAgent and EXAgent have almost the same performance, because they selected the right explanation (*skeleton-attacked*) in the beginning and generate a goal to deal with the external event.

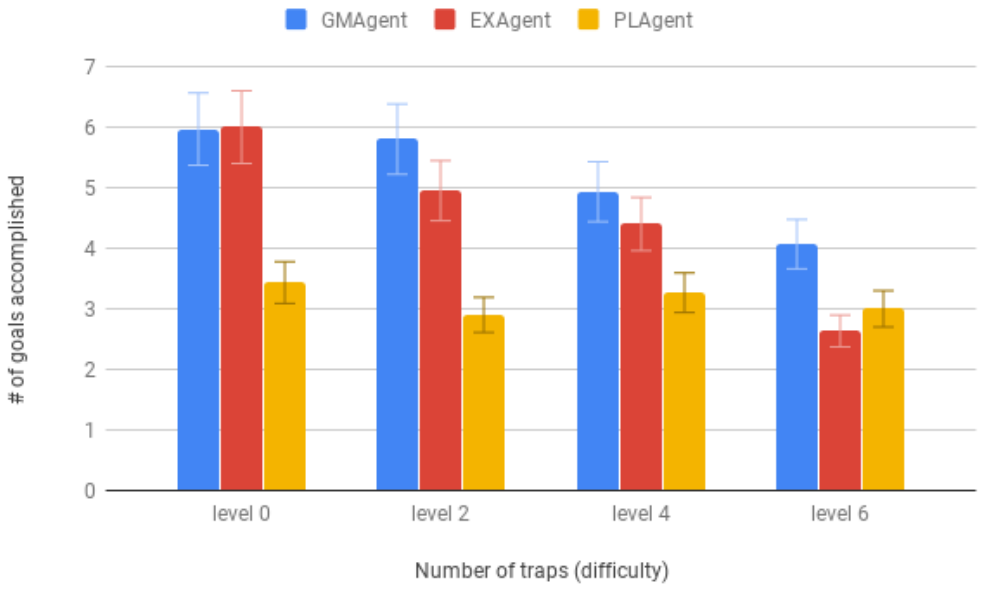

Figure 6.3: The performance of the GMAgent, EXAgent and PLAgent in the Minecraft domain. Difficulty is based on the number of traps (i=0,2,4,6). The chart shows the average of 50 runs at each difficulty level.

When there are more traps, the GMAgent has a better overall performance score than the EXAgent. In the middle of plan execution a monitor fires if the GMAgent did not find a skeleton or if it receives a new observation that proves there is an arrow trap nearby. The agent responds by changing the goal to destroy the trap instead of the skeleton. In contrast, the EXAgent persists in pursuit of the goal (kill skeleton), but it dies before it gets to the real problem.

At difficulty level six, the performance of PLAgent is slightly better than EXAgent, because the EXAgent wastes time pursuing the wrong goal, which causes damage. The PLAgent ignores the skeleton and trap and pursues the original goal. This comparison shows that the GMAgent makes better progress towards achieving its goals compared to the EXAgent and PLAgent which supports our claim that using goal monitors improves the agent's performance.

Related Research

The related research is organized as follows: goal reasoning agents, which are able to reason and adjust their goals; cognitive agents, that integrate act, planning, and perception such as SOAR and ICARUS; planning systems, which describe agents that generate a plan to achieve goals and integrate planning and execution to increase the robustness.

7.1 Goal Reasoning Agents

Goal reasoning [71] agents can extend their reasoning capabilities beyond their plans and actions to consider their goals [46]. Goal reasoning studies how autonomous agents can reason about and adjust their goals. Goal-driven autonomy (GDA) [3, 20, 48] is a kind of goal-reasoning that focuses on the explanation of discrepancies in order to formulate new goals. Our work is firmly situated within this research area. GDA separates the planning process from procedures for goal formulation and management. GDA agents generate goals as the agent encounters differences between the agent's expectations for the outcome of its actions and the actual observed outcomes in each new state [26]. When such a discrepancy occurs, GDA agents generate a causal explanation for the discrepancy and reformulate a new goal based on the causal structure.

Research on goal formulation and goal management dates back to the work by Bratman [12] on the introduction of the BDI (belief-desire-intention) model. ARTUE uses an engineered rule set that formulates goals when its trigger conditions are met. This ap-

proach is limiting because the agent only knows to respond to situations that the domain designer anticipates. Other agents have a more robust goal selection approach through the use of learning. EISBot [74] learns goal selection knowledge from human demonstrations. Learning GDA [45] learns its goal selection function using Q-learning. M-ARTUE [76] is a domain-independent autonomous agent with the capacity to dynamically determine which goals to pursue in unexpected situations. M-ARTUE performs goal formulation using domain-independent heuristics. This allows the agent to react robustly to unexpected changes. There are three motivators to guide the agent's behavior: *Social Motivator* which encodes the desire to pursue the goals provided by an external agent, *Opportunity Motivator* which chooses the goals to gather and conserve resources, and *Exploration Motivator* which maximize the agent's opportunities to explore new states. M-ARTUE selects goals based on two metrics that are calculated from how urgent the motivator is and how well a plan fulfills the motivator. Based on these metrics, the agent then can decide which goal is a higher priority to achieve during execution. For example, the metric value for an exploration motivator is initially high to encourage the agent to gather more information in the beginning, and the metric value for the Opportunity motivator increases when the agent expands its resources. This work focuses on the problem of goal selection using domain-independent metrics, but it does not discuss goal monitoring or goal change due to change in the world that affects the selected goals. Our approach enables an agent to know when the goal is not necessary in the current state.

Roberts et al. [69, 67] have formalized the process of goal change and goal reasoning. They developed a notation for a goal life cycle where goals transition through modalities that represent goal formulation, goal selection, goal expansion, goal commitment, goal dispatching, goal monitoring, goal evaluation, goal repair, and goal deferment. Many of these transitions correspond to our goal operations, but their formalism treats goal reasoning as goal refinement where ours casts it in the language of goal operations. Additionally Roberts et al. propose a complex goal structure that differs from ours. Their goal node

includes not only the desired state but also super-ordinate and subordinate goal linkages, goal constraints, quality metrics, and pointers to the current plan associated with the node.

Coddington [17] introduced MADbot that generates goals in response to changes during planning and execution. MADbot is a planning and execution architecture for a motivated, autonomous planning agent. The role of a motivation system is to direct an agent towards one of its different tasks. They describe two approaches to goal generation. In the first model of goal generation, goals are generated explicitly in response to changes that occur to the agents motivations. This approach is external to the planner and the planner does not know about the resource consumption. The second one models the motivations internally within the planner's domain. They show that the combination of these two models can be the best approach. In this hybrid model, those motivations which are resource-critical are modeled as resources in the planner. Those motivations that are not resource-critical are explicitly modeled as motivations. These monitor the value of their associated state variables and generate goals whenever their motivational values satisfy the appropriate constraints. This work is about goal generation and not reasoning about goals after they are generated as we propose to do. Also, the goals are not monitored after they are added to the system.

Rebel agents is a new research area in goal reasoning [18, 10] that refers to agents that can object to or completely reject the goals they are given by the external agents. Aha and Coman [18] introduce a framework to enable discussion, implementation, and deployment of positive rebel agents. This framework enables the agent to change its attitude toward a goal due to environmental change or new knowledge about the operator's behavior. In this work [10], the agent rejects goals with undesirable effects.

The BDI community developed a goal life cycle representation and formalized a set of goal operations [38]. This work, unlike the goal reasoning community, does not focus as much on goal change and goal formulation. The BDI community also developed a sophisticated mechanism for performing goal suspension, resumption, and abandonment

[39]. Their work differentiates goal abandonment (where plans are cleaned up and then the goal is dropped) from the direct goal drop operation itself. The BDI work cited above characterizes a kind of goal monitoring for maintenance goals that assure a particular state holds across an interval of time. These goals contrast with achievement goals that establish a particular state at a point in time. This process monitors the state and (re)activates the maintenance goal whenever the state changes during the interval. Our use of goal monitoring is directed at achievement (i.e., attainment) goals and monitors the reasons goals were formulated in the first place. Our work anticipates what changes affect the goal, we let the monitors observe them and if they change, then the agent may drop or change the goal.

Wooldridge [78] provides a model of cooperative problem solving, in which some agents cooperate to achieve a common goal. They address the issue when the agents abandon their joint commitment of achieving a shared goal. Joint commitments are held by the agents corresponding to a motivation, which includes the justification for the commitment. When the motivation/justification for the goal is no longer present the agents terminate the joint commitment. This work is similar to our goal monitoring approach when the justification of a goal is not valid and the agent abandons the goal.

7.2 Cognitive Architectures

Much of the prior work in AI regarding interleaving planning and interpretation has been carried out in the form of cognitive architectures. Work on monitoring has mainly been done in individual systems (e.g. HOTRIDE). Our work integrates planning and interpretation through monitoring as we do both plan monitoring and goal monitoring. We not only use monitors in execution, but we use monitors during plan generation. These plan monitors are particularly useful when planning takes a considerable amount of time. In real-world examples (e.g., large-scale military logistics scheduling), planning duration can be extensive, so significant changes often occur in the interval. We also integrate our system

with an actual robot using a particular vision system. As far as we are aware, no previous work implemented all of these capabilities in one system. Specifically, we treat perception as an active process that is guided by other cognitive components such as planning. This is in contrast to a visual system that once initialized does not change its overall behavior or focus. In most previous work, perception simply provides labeled data of a scene, and cognitive processes make do with whatever information is output from perception.

One important aspect of this work is related to cognitive architectures. Cognitive architectures integrate many capabilities associated with human intelligence. These capabilities are interacting with dynamic complex environments, pursuing a wide variety of tasks, using a large body of knowledge, and planning and learning from experience.

Systems like ICARUS, SOAR, and MIDCA are inspired by human cognition and are designed to accomplish multiple tasks. The ICARUS architecture [52] includes modules for conceptual inference, goal selection, skill execution, means-ends problem solving, and skill learning. The Inference module is bottom-up processing of perceptual input with rules that fire in certain circumstances. Concept definitions and a set of skills are stored in long-term memory for each domain. Short-term memories store the instantiated concepts for the current situation which are called *beliefs*. Also, instantiated skills in the form of special constructs that include a goal and a set of satisfied preconditions are stored in the memories too. Basically, there are rules with a set of preconditions that when valid in the state generates a goal.

ICARUS operates in cycles. In each cycle, it perceives the environment, infers beliefs and goals from percepts. It tries to match the percepts against the concepts in long-term memory and extract beliefs. It has a bottom-up inference process to build all the primitive and non-primitive concept instances for the current situation. ICARUS examines the rules at each cycle to generate goals. Then it prioritizes each goal and selects one to pursue.

A new extension to ICARUS's architecture [16] allows reactive goal management. This feature enables the agent to change the priority of goals dynamically based on the

new state. They implement a metric that changes based on the situation and can influence the goal priorities. It also has the ability to retract the goal if the situation changes and an event that triggered a goal disappears. This is similar to the operator-based goal monitor. Although, we have implemented Explanatory goal monitors which extract the reasons why the agent pursues a goal from an explanation of a discrepancy and perform monitoring of those reasons.

The PUG extension to the ICARUS cognitive architecture [53] is the closest work to our MIDCA architecture in spirit. PUG integrates action execution, planning, and plan monitoring. The monitors check for the preconditions and effects of actions during execution, and if they find any anomaly (e.g. the preconditions of any action are not satisfied) it leads to replanning. They also monitor the preconditions of the rules that generate goals in PUG, and if the conditions change, the goal will be abandoned. Our work is related to their approach in that we address the problem of plan execution and goal monitoring. However, the nature of our explanation-based goal monitors is different since they are concerned with goals generated from dynamic explanatory structures rather than static rules. Additionally in our work, the response to changes is not only a choice between goal abandonment or replanning; our agent may change the goal itself in addition to simply dropping it.

SOAR [51] organizes behavior as a search through a problem space; each cycle uses knowledge to add elements to working memory that helps it select operators to carry out. Laird and Rosenbloom [51] report a version of Soar that senses an external environment, carries out physical actions, and integrates planning with execution. This extension of Soar revises the plan in response to changes during planning and execution time. For example, in the blocksworld domain, while the robot is planning to align the blocks one of the blocks is removed from the table and the operators related to the removed blocks will be retracted from the memory.

Soar's SVS (Spatial/Visual System) [50, 55] makes use of visual information for problem-solving and planning. SVS is designed based on human visual imagery. The

memories in SVS contain non-symbolic representations of the world (i.e., spatial and visual information). The process of requesting predicates from working memory is called querying SVS. The predicates are extracted from image data only when requested by Soar. The agent can dynamically query the scene for features/relationships of interest to the task. Soar implements a top-down visual recognition since the agent creates symbolic commands based on the problem. In our system, the monitors which are automatically generated during planning specify what predicates to extract from the image data.

Heigh and Veloso [37] report on the ROGUE system, an extension to the Prodigy architecture [14] that deals with unexpected events and the insertion of new goals while planning. ROGUE integrates the tasks of planning, execution and learning for a mobile robot that accepts asynchronous requests from users. This system deals with unexpected events and insertion of new goals. ROGUE has two types of monitors: *Action monitors* which monitor the preconditions and effects of the actions and *environment monitors* which observe the exogenous events in the environment that can affect the goals (e.g., monitoring the battery power). Environment monitors are specified by the programmer. Environment monitors' purpose is to realize new opportunities. Both monitors result in replanning. ROGUE's monitoring algorithms determine which information is relevant for planning and replanning. If an action deachieves the effect of a previous action, ROGUE adds a subgoal to a set of pending goals. Instead of replanning from scratch, it adds or deletes steps from/to the plan [36].

7.3 Planning

Classical planning ignores the uncertainty of the world [31] and uses a deterministic planner to generate a plan. When the execution fails, replanning is performed. These pioneering approaches have been successful in some domains but they are criticized for lack of reactivity, and lots of work has been done to handle the uncertainty of the world. Replanning

algorithms such as FF-replan [80], invoke an offline planner with a deterministic domain, and the plan is executed until the planner observes an unexpected outcome. At this point, the planner is called again. The classical planners assume that they have complete knowledge about the world, and they don't make any effort to obtain information.

7.3.1 Integrated planning and execution

The AI planning community has developed systems that integrate planning, execution and monitoring. Several model-based executives interleave planning and execution to improve robustness. For instance, IxTeT [56] dynamically produced a partial plan during planning phase and executed and repaired the plan during the act phase; PRS [34] used domain knowledge to generate hierarchical plans, execute them in the environment, and monitor their progress. It also used other knowledge to change goals in response to unexpected events; Wilkins' SIPE (System for Interactive Planning and Execution Monitoring) [75] and IPEM (Integrated Planning, Execution, and Monitoring) [5] initiate an error recovery during execution when something has gone wrong; HOTRiDE [7] modifies the plan during execution using a dependency graph; and CASPER [6] refines the plan under construction when the external information changes during planning time.

HOTRiDE [7] performs action monitoring in a simulated domain and repairs the plan if any action fails. After executing each action, the new state is observed. To execute a new action, the controller checks to see if all the preconditions of the action are satisfied in the state. If not, the execution stops, and the controller calls HOTRiDE to replan from the failure point regarding the new state. It uses a dependency graph to find which task decompositions are not valid in the current state and need replanning. This dependency graph keeps the information about the dependency among tasks decomposed by the SHOP planner [62] during planning. When an action fails, using this information, the planner knows which decompositions are not valid, so it knows where exactly to backtrack to refine the plan.

To refine the plan, HOTRiDE checks every parent task of the failed action to find a task which can be decomposed in another way. If there is no such a task, HOTRiDE returns failure. Also using the dependency graph, it will find all other related actions of the failed task. The new decomposition should support those actions too. The repaired plan might include the actions that were executed before in the failed plan. Their approach can not identify the redundant/duplicate actions. This system only focuses on the changes during execution.

Another system, CASPER [6], refines the plan under construction when the external information changes during planning time. This approach is useful for planning in situations where the planner needs to get information from external sources where the information may change during plan generation. They use query management strategies that can adapt existing planners to deal with volatile external information by backtracking the planner to the first point in the planning process. Our approach generates monitors during planning dynamically instead of issuing predefined queries at a certain time.

Unfortunately, most of these planning systems have very complex architectures compared to other AI planning systems. While many of these strategies focus on repairing the plan when observations indicate that it cannot succeed, the planning techniques employed by these systems can be time-consuming when replanning is frequently necessary. Also, None of these systems perform monitoring on explicit goal structures.

7.3.2 Contingent Planning

Contingent planning addresses the planning problem with the uncertainty of the initial state and action effects. It can be translated to search in a space of possible worlds. The number of possible worlds in a belief state can be large in more complex domains which makes these planners hard to scale up. Some contingent planners include Contingent-FF [42], POND [13], and MBP [8].

The MBP planner [8] generates conditional plans that branch on conditions of the ob-

servable variables' value. After performing a sensing action, multiple possible values for an observation may occur. MBP allows the user to specify the observation values that can be observed during execution. MBP allows for both observations resulting from the execution of sensing actions and automatic sensing that depends on the current state of the world. Shani and Brafman [70] describe the SDR planner which extends the replanning algorithm to contingent domains with partial observability and uncertainty about the initial state. This method generates a problem that reasons about the agent's state of knowledge rather than the state of world. It selects a sample of possible states to reduce the complexity, and then uses a method of regression at each step of the plan to make sure the new observation is consistent with the assumed initial state. If it is not, then the planner changes the set of possible initial states. When the SDR planner senses information that contradicts its beliefs, it gives up the current plan and generates a new sample of possible initial states and replans.

7.3.3 Execution Monitoring

The outcome of executing a plan is not always as expected during planning. Lots of works have studied execution monitoring frameworks that address this problem. Our work is related to the problem of execution monitoring; the problem of detecting failures during execution and recovering from them [29, 5].

Expectation-based monitors detect failures in execution by comparing expectations generated by the planner to observations received during execution. Mendoza and Veloso [58] focus on monitoring stochastic expectations and finding subtle anomalies from collections of observations, and adapting the model to improve performance based on experience. This work improves the world model using past experience for future plan execution. This approach create a plan with a sequence of actions and a set of expectations based on the effect of actions. During execution, it monitors to see if the expectations are met. It then finds sets of conditions in which observations deviate from expectations and based on these conditions, it modifies its model of the world.

Similar work to the perceptual plan monitors has been previously performed by Veloso, Pollack and Cox [73], who implemented rationale-based monitors in the state-based Prodigy planning system. Our approach differs in using these monitors in the SHOP HTN planner. They also do not have some of the main components of our work, such as goal monitoring, reasoning or using the perceptual component in plan monitors.

Yezhou [79] introduces an architecture, DeepIU, which is able to understand an image, answer questions about it, reason about the content, and update the agent's belief about the image. This architecture allows a cycle of vision-reasoning-vision and reasoning-vision-reasoning-vision. This ability allows iteration through vision and reasoning modules to achieve complete understanding of the image. If more detail is needed, the reasoning module outputs a possible question that makes the visual detection module process a specific region of the image. The Scene Description Graph (SDG) is a knowledge structure that captures information from the vision and reasoning modules. It is a directed graph among entities, events and their properties that can be used to answer questions about the image. The SDG is a representation of the scene that integrates information received from the visual module (e.g., objects and their properties) with background knowledge. To generate SDGs, DeepIU does the following three things:

1. DeepIU collects information for object classes and scene classes. For scene classes, it keeps synonyms and a list of ASCs (abstract scene constituents) which describes that scene.

2. Then it creates a knowledge base and a Bayesian network to capture commonsense knowledge about the domain. This architecture uses a K-parser to extract entities, traits and events. The Bayesian network represents the dependencies among the entities.

3. At the end, using the knowledge base and detected entities from the visual node, SDG is constructed for an image.

The DeepIU reasoning module helps to have a better understanding of the image, and it guides vision to search “what and where” in the image. Our system guides vision based on the agent’s goal and plan and helps it to know what to look at the image.

Conclusions

In this dissertation, we presented a novel approach of integrating perception, action, planning and interpretation. We introduced anticipatory plan monitors and goal monitors as tools for an autonomous agent to be more robust to changes in a dynamic world. These monitors enable the agent to anticipate which features are related to its goals and plans; monitor these features; and respond appropriately if these features change due to external events via plan adaptation or goal change. This framework enables the agent to be more flexible and reactive which is necessary to cope with a changing environment.

We have argued that perception should serve the needs of the planner. The planner generates perceptual monitors for the underlying vision system based on the rationale for plan decisions (e.g., preconditions), and the perceptual system detects when these conditions are violated. Our results support the notion that perception has an important role in supporting the intentions and actions of the agent. These results promise significant methods for handling complexity and change across the range of problems autonomous agents may encounter in the real world.

Perception also serves the needs of interpretation with goal monitors. Given a particular scene or situation, MIDCA's interpretation phase can recognize new problems in terms of expectation failures or discrepancies. The interpretation system will then attempt to explain the discrepancy using an environment model. It then formulates goals and monitors the justification for goal selection and abandons/changes the goal when justifications are not valid in the environment. Each goal monitor identifies the conditions under which

the goal should be reconsidered, and also describes how the agent should behave after theses conditions change. In addition, we reported our system's operation on scenarios that demonstrated its ability to handle the situation when there are multiple hypotheses about the world and switch goals if they are believed to be true or false.

8.1 Status of Claims

Our first claim stated that perception, action, planning and interpretation are dependent. In Chapter 3 and 4, we showed how these cognitive processes interact using plan monitors and goal monitors. In Chapter 5, we showed that perceptual plan monitors are key when integrating planning and interpretation. We examined how plan monitors help planning and interpretation, and conversely how planning and interpretation help perception. In Section 5.3, we showed how planning provides focus to perception and reduces the number of objects and relations that need to be processed.

Our second claim stated that using plan monitors improves the planning performance of an agent in complex domains. In Chapter 5, we reported results from experimental scenarios regarding the performance of plan monitors. Our results showed that plan monitors reduce the planning cost of a cognitive agent in a dynamic world.

Our third claim stated that using goal monitors would increase the number of goals that a cognitive agent can achieve. In Section 6.2, we showed the performance of operator-based goal monitors in a modified logistics domain and compared its performance with a cognitive agent with no goal monitors. The results showed that using goal monitors enable the agent to achieve more goals in a dynamic environment. In Section 6.3, we defined two baseline agents EXAgent, a GDA agent and PLAgent, an offline planning agent, and compared them experimentally with GMAgent (our GDA agent with goal monitors), showing that in a partially observable domain, GMAgent achieved more goals than the two other baseline agents.

8.2 Future Work

There are many promising areas of future work stemming from this research. We consider some of them in the following paragraphs.

Confidence measures for derived hypotheses: Building upon explanatory goal monitors, future work should investigate reasoning over pieces of evidence to calculate the probability for each hypothesis. When new information is received the system should reason about it before deciding about dropping or changing its current goal. By providing a probability for each piece of evidence, we can use Bayesian probability theory to calculate the probability of each event and let the goal monitor decide if the goal change is needed when it observes new pieces of evidence.

Goal monitors for a multi-agent system: The research in this thesis focuses on single agent systems. We would like to extend the use of goal monitors for cooperative problem solving, in which a group of agents are collaborating to achieve a shared goal [78]. Woodbridge discussed justifications for goals that cease to hold when the agents abandon their commitment to achieve shared goals. We would like to explore the use of the explanatory goal monitors in these cooperative systems. The agents should also share their knowledge about the world. Different agents may explain a situation differently since they have a different view of the world. If the agents share their hypotheses, then they can agree on one explanation. If any of the agents observe new evidence, then they can refine the explanation and the goal associated with it.

Active Interpretation of Disparate Alternatives[1]: The AIDA project (DARPA, ongoing) is to design an engine that generates explicit alternative hypotheses of events and situations from a variety of unstructured sources. We would like to explore the use of our goal monitor approach for such a system to handle competing explanations for a situation.

Relationship with work on expectations: We want to study the relationship between our work and informed expectation [26]. Informed expectation accumulates the effects of

[1] https://www.darpa.mil/program/active-interpretation-of-disparate-alternatives

actions and checks at each state if the effects are still valid in the current state. This work is useful in the GDA process to detect an anomaly. Our plan monitors check the preconditions to make sure they are valid in the current state and if not refine the plan under construction. Monitors distinguish between the changes that effect the goal and plan and respond based on how these feature effect the agent's behavior.

Adding a Reasoning Component to the Monitors: Some situations warrant responses after a monitor fires, whereas other situations lack the immediacy that demands a full response of change to either the plan or the goal. We will extend the representation of perceptual monitors to include a reasoning step after the monitor fires to verify the response if necessary.

Relationship with work on metacognition The goal-related processes like monitoring have inherent aspects of metacognition in them. Since the goal monitors provide the identification for a goal change, we would like to explore how this goal monitoring is related to metacognition. We would like to merge this work with previous research on goal transformation [22]. We would like to use monitors to identify situations that require goal change and decide what kind of goal transformation is needed in response, and how metacognition can be used in this process.

When autonomous agents operate in the real world, execution failure is inevitable. The contributions of this thesis are important because the monitors make cognitive agents more robust to changes in a dynamically changing world. The monitors enable the agents to manage their behavior across many complex situations and to solve severe problems that arise while pursuing their goals. This research promises significant methods for handling complexity and change across the range of problems autonomous agents may encounter in the real world.

Bibliography

[1] David W Aha. Goal reasoning: Foundations, emerging applications, and prospects. *AI Magazine*, 39(2):3–24, 2018.

[2] David W Aha, Tory S Anderson, Benjamin Bengfort, Mark Burstein, Dan Cerys, Alexandra Coman, Michael T Cox, Dustin Dannenhauer, Michael W Floyd, Kellen Gillespie, et al. *Goal Reasoning: Papers from the ACS workshop*. Technical report, Georgia Institute of Technology, 2015.

[3] DW Aha, M Klenk, H Munoz-Avila, A Ram, and D Shapiro. *Goal-driven autonomy: Notes from the AAAI workshop*, 2010.

[4] Zohreh Alavi and Michael T Cox. Rational-based visual planning monitors. In *Proceedings of the 4th Workshop on Goal Reasoning, IJCAI-16*, pages 3968–3969, 2016.

[5] Jose A Ambros-Ingerson and Sam Steel. Integrating planning, execution and monitoring. In *AAAI Conference on Artificial Intelligence*, volume 88, pages 21–26. AAAI Press, 1988.

[6] Tsz-Chiu Au, Dana Nau, and VS Subrahmanian. Utilizing volatile external information during planning. In *Proceedings of the 16th European Conference on Artificial Intelligence*, pages 647–651. Citeseer, 2004.

[7] N Fazil Ayan, Ugur Kuter, Fusun Yaman, and Robert P Goldman. Hotride: Hierarchical ordered task replanning in dynamic environments. In *Planning and Plan Execution for Real-World Systems–Principles and Practices for Planning in Execution: Papers from the ICAPS Workshop. Providence, RI*, volume 38, 2007.

[8] Piergiorgio Bertoli, Alessandro Cimatti, Marco Pistore, Marco Roveri, and Paolo Traverso. MBP: a model based planner. In *Proceedings of the IJCAI01 Workshop on Planning under Uncertainty and Incomplete Information*, 2001.

[9] Avrim L Blum and Merrick L Furst. Fast planning through planning graph analysis. *Artificial intelligence*, 90(1-2):281–300, 1997.

[10] James Boggs, Dustin Dannenhauer, Michael W Floyd, and David Aha. The ideal rebellion: Maximizing task performance in rebel agents. In *6th Goal Reasoning Workshop at IJCAI/FAIM-18*, 2018.

[11] SRK Branavan, Nate Kushman, Tao Lei, and Regina Barzilay. Learning high-level planning from text. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics: Long Papers-Volume 1*, pages 126–135. Association for Computational Linguistics, 2012.

[12] Michael Bratman. *Intention, plans, and practical reason*, volume 10. Harvard University Press Cambridge, MA, 1987.

[13] Daniel Bryce, Subbarao Kambhampati, and David E Smith. Planning in belief space with a labelled uncertainty graph. In *AAAI Workshop on Learning and Planning in Markov Decision Processes*. AAAI Press, 2004.

[14] Jaime Carbonell, Oren Etzioni, Yolanda Gil, Robert Joseph, Craig Knoblock, Steve Minton, and Manuela Veloso. Prodigy: An integrated architecture for planning and learning. *ACM SIGART Bulletin*, 2(4):51–55, 1991.

[15] Kwai-Cheung Chan. *Operation Desert Storm: Evaluation of the air campaign*. Diane publishing, 1997.

[16] Dong Kyu Choi. *Coordinated execution and goal management in a reactive cognitive architecture*. PhD thesis, Stanford University, 2010.

[17] Alexandra Coddington. Motivations for MADbot: a motivated and goal directed robot. In *Proceedings of the Twenty-Fifth Workshop of the UK Planning and Scheduling Special Interest Group*, pages 39–46, 2006.

[18] Alexandra Coman, Kellen Gillespie, and Héctor Muñoz-Avila. Case-based local and global percept processing for rebel agents. In *International Conference on Case-Based Reasoning(ICCBR) (Workshops)*, volume 1520, pages 23–32, 2015.

[19] Michael T Cox. Perpetual self-aware cognitive agents. *AI Magazine*, 28(1):32, 2007.

[20] Michael T Cox. Goal-driven autonomy and question-based problem recognition. In *Second Annual Conference on Advances in Cognitive Systems 2013, Poster Collection*, pages 29–45. Palo Alto, CA: Cognitive Systems Foundation, 2013.

[21] Michael T Cox, Zohreh Alavi, Dustin Dannenhauer, Vahid Eyorokon, Hector Munoz-Avila, and Don Perlis. MIDCA: A metacognitive, integrated dual-cycle architecture for self-regulated autonomy. In *In Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, Vol. 5 (pp. 3712-3718)*. AAAI Press, 2016.

[22] Michael T Cox, Dustin Dannenhauer, and Sravya Kondrakunta. Goal operations for cognitive systems. In *Thirty-First AAAI Conference on Artificial Intelligence*. AAAI Press, 2017.

[23] Michael T Cox and Zohreh A Dannenhauer. Perceptual goal monitors for cognitive agents in changing environments. In *Proceedings of the Fifth Annual Conference on Advances in Cognitive Systems, Poster Collection*, pages 1–16, 2017.

[24] Michael T Cox and Ashwin Ram. On the intersection of story understanding and learning. In *Understanding language understanding*, pages 397–433. MIT Press, 1999.

[25] Dustin Dannenhauer, Michael W Floyd, Daniele Magazzeni, and David W Aha. Explaining rebel behavior in goal reasoning agents. In *Proceedings of ICAPS-18 Workshop on Explainable Planning*, 2018.

[26] Dustin Dannenhauer and Hector Munoz-Avila. Raising expectations in gda agents acting in dynamic environments. In *International Joint Conference on Artificial Intelligence (IJCAI-15)*, 2015.

[27] Zohreh Dannenhauer and Michael Cox. Rationale-based perceptual monitors. *AI Communications Journal*, 31(2):197–212, 2018.

[28] Zohreh A Dannenhauer and Michael T Cox. Rational-based visual planning monitors for cognitive systems. In *Proceedings of the 30th International FLAIRS Conference*, pages 182–185. AAAI Press, 2017.

[29] Giuseppe De Giacomo, Raymond Reiter, and Mikhail Soutchanski. Execution monitoring of high-level robot programs. In *Principle of knowledge and representation and reasoning conference*, pages 453–465. Morgan Kaufman publishers, 1998.

[30] Cornelia Fermüller and Yiannis Aloimonos. Vision and action. *Image and vision computing*, 13(10):725–744, 1995.

[31] Richard E Fikes, Peter E Hart, and Nils J Nilsson. Learning and executing generalized robot plans. *Artificial intelligence*, 3:251–288, 1972.

[32] Richard E Fikes and Nils J Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1971.

[33] Maria Fox and Derek Long. PDDL2. 1: An extension to PDDL for expressing temporal planning domains. *Journal of artificial intelligence research*, 20:61–124, 2003.

[34] Michael P Georgeff and Amy L Lansky. Reactive reasoning and planning. In *AAAI Conference in Artificial Intelligence*, volume 87, pages 677–682. AAAI Press, 1987.

[35] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated planning: Theory & practice*. Elsevier, 2004.

[36] Karen Zita Haigh and Manuela M Veloso. Using perception information for robot planning and execution. In *Proceedings of the AAAI Workshop\Intelligent Adaptive Agents*, pages 23–32, 1996.

[37] Karen Zita Haigh and Manuela M Veloso. Interleaving planning and robot execution for asynchronous user requests. *Autonomous Robots*, 5(1):79–95, 1998.

[38] James Harland, David N Morley, John Thangarajah, and Neil Yorke-Smith. An operational semantics for the goal life-cycle in BDI agents. *Autonomous Agents and Multi-agent Systems*, 28(4):682–719, 2014.

[39] James Harland, David N Morley, John Thangarajah, and Neil Yorke-Smith. Aborting, suspending, and resuming goals and plans in BDI agents. *Autonomous Agents and Multi-Agent Systems*, 31(2):288–331, 2017.

[40] Nick Hawes. A survey of motivation frameworks for intelligent systems. *Artificial Intelligence*, 175(5-6):1020–1036, 2011.

[41] Jörg Hoffmann. The Metric-FF planning system: Translating "ignoring delete lists" to numeric state variables. *Journal of Artificial Intelligence Research*, 20:291–341, 2003.

[42] Jörg Hoffmann and Ronen Brafman. Contingent planning via heuristic forward search with implicit belief states. In *International Conference on Automated Planning and Scheduling (ICAPS)*, volume 2005, 2005.

[43] Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[44] Michael N Huhns and Munindar P Singh. Cognitive agents. *IEEE Internet computing*, 2(6):87–89, 1998.

[45] Ulit Jaidee, Héctor Muñoz-Avila, and David W Aha. Integrated learning for goal-driven autonomy. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, volume 22, page 2450, 2011.

[46] Benjamin Johnson, Michael W Floyd, Alexandra Coman, Mark A Wilson, and David W Aha. Goal reasoning and trusted autonomy. In *Foundations of trusted autonomy*, pages 47–66. Springer, 2018.

[47] Randolph M Jones, John E Laird, Paul E Nielsen, Karen J Coulter, Patrick Kenny, and Frank V Koss. Automated intelligent pilots for combat flight simulation. *AI Magazine*, 20(1):27, 1999.

[48] Matthew Klenk, Matt Molineaux, and David W Aha. Goal-driven autonomy for responding to unexpected events in strategy simulations. *Computational Intelligence*, 29(2):187–206, 2013.

[49] Sravya Kondrakunta and Michael T Cox. Autonomous goal selection operations for agent-based architectures. In *Working Notes of the 2017 IJCAI Goal Reasoning Workshop*. IJCAI, 2017.

[50] John E Laird, Keegan R Kinkade, Shiwali Mohan, and Joseph Z Xu. Cognitive robotics using the Soar cognitive architecture. *Cognitive Robotics AAAI Technical Report WS-12-06*, pages 46–54, 2012.

[51] John E Laird and Paul S Rosenbloom. Integrating, execution, planning, and learning in Soar for external environments. In *Proceedings of the eighth AAAI conference on artificial intelligence*, volume 90, pages 1022–1029, 1990.

[52] Pat Langley and Dongkyu Choi. A unified cognitive architecture for physical agents. In *Proceedings of the twenty-first AAAI conference on artificial intelligence*. AAAI Press, 2006.

[53] Pat Langley, Dongkyu Choi, Mike Barley, Ben Meadows, and Edward P Katz. Generating, executing, and monitoring plans with goal-based utilities in continuous domains. In *Proceedings of the Fifth Annual Conference on Advances in Cognitive Systems*, 2017.

[54] Pat Langley, John E Laird, and Seth Rogers. Cognitive architectures: Research issues and challenges. *Cognitive Systems Research*, 10(2):141–160, 2009.

[55] Scott D Lathrop, Samuel Wintermute, and John E Laird. Exploring the functional advantages of spatial and visual cognition from an architectural perspective. *Topics in cognitive science*, 3(4):796–818, 2011.

[56] Solange Lemai and Félix Ingrand. Interleaving temporal planning and execution in robotics domains. In *AAAI Conference on Artificial Intelligence*, volume 4, pages 617–622. AAAI Press, 2004.

[57] David Marr. A computational investigation into the human representation and processing of visual information. *Vision*, pages 125–126, 1982.

[58] Juan Pablo Mendoza, Manuela Veloso, and Reid Simmons. Plan execution monitoring through detection of unmet expectations about action outcomes. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3247–3252. IEEE, 2015.

[59] Matthew Molineaux. *Understanding what may have happened in dynamic, partially observable environments*. PhD thesis, George Mason University, 2017.

[60] Matthew Molineaux, Matthew Klenk, and David W Aha. Goal-driven autonomy in a navy strategy simulation. In *AAAI Conference on Artificial Intelligence*, pages 1548–1554, 2010.

[61] Matthew Molineaux, Ugur Kuter, and Matthew Klenk. DiscoverHistory: Understanding the past in planning and execution. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*, pages 989–996. International Foundation for Autonomous Agents and Multiagent Systems, 2012.

[62] Dana Nau, Yue Cao, Amnon Lotem, and Héctor Muñoz-Avila. "SHOP": Simple hierarchical ordered planner. In *Proceedings of the 16th IJCAI-Vol. 2*, pages 968–973. Morgan Kaufmann, 1999.

[63] Dana S Nau. Current trends in automated planning. *AI Magazine*, 28(4):43–43, 2007.

[64] Matt Paisner, Michael Maynord, Michael T Cox, and Don Perlis. Goal-driven autonomy in dynamic environments. In *Goal Reasoning: Papers from the ACS Workshop*, pages 79–94. Tech. Rep. No. CS-TR-5029. College Park, MD: University of Maryland, Department of Computer Science, 2013.

[65] Matthew Paisner, Michael Cox, Michael Maynord, and Don Perlis. Goal-driven autonomy for cognitive systems. In *Proceedings of the Annual Meeting of the Cognitive Science Society*, volume 36, pages 2085 – 2090. Austin, TX: Cognitive Science Society, 2014.

[66] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. ROS: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, 2009.

[67] Mak Roberts, Vikas Shivashankar, Michael Alford, Ron; Leece, Shubham Gupta, and David Aha. Goal reasoning, planning, and acting with actorsim, the actor simulator. In *Poster Proceedings of the Fourth Annual Conference on Advances in Cognitive Systems*. Evanston, IL, USA, 2016.

[68] Mark Roberts, Daniel Borrajo, Michael Cox, and Neil Yorke-Smith. Special issue on goal reasoning. *AI Communications Journal*, 31(2):115–116, 2018.

[69] Mark Roberts, Swaroop Vattam, Ronald Alford, Bryan Auslander, Tom Apker, Benjamin Johnson, and David W Aha. Goal reasoning to coordinate robotic teams for disaster relief. In *Proceedings of ICAPS-15 PlanRob Workshop*, pages 127–138, 2015.

[70] Guy Shani and Ronen Brafman. Replanning in domains with partial information and sensing actions. In *Twenty-Second International Joint Conference on Artificial Intelligence*, 2011.

[71] Swaroop Vattam, Matthew Klenk, Matthew Molineaux, and David W Aha. *Breadth of approaches to goal reasoning*: A research survey. Technical report, Naval Research Lab Washington DC, 2013.

[72] Manuela M Veloso. *Planning and Learning by Analogical Reasoning*. Springer Verlag, 1994.

[73] Manuela M Veloso, Martha E Pollack, and Michael T Cox. Rationale-based monitoring for planning in dynamic environments. In *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems (pp. 171-179). Menlo Park, CA*, pages 171–180. AAAI Press, 1998.

[74] Ben George Weber, Michael Mateas, and Arnav Jhala. Learning from demonstration for goal-driven autonomy. In *Twenty-Sixth AAAI Conference on Artificial Intelligence*. AAAI Press, 2012.

[75] David E Wilkins. Recovering from execution errors in sipe. *Computational Intelligence*, 1(1):33–45, 1985.

[76] Mark A Wilson, Matthew Molineaux, and David W Aha. Domain-independent heuristics for goal formulation. In *The Twenty-Sixth International FLAIRS Conference*, 2013.

[77] Terry Winograd. Procedures as a representation for data in a computer program for understanding natural language. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE PROJECT MAC, 1971.

[78] Michael Wooldridge and Nicholas R Jennings. The cooperative problem-solving process. *Journal of Logic and computation*, 9(4):563–592, 1999.

[79] Yezhou Yang, U EDU, Aloimonos Y, and Cornelia Fermuller. DeepIU: An architecture for image understanding. *Advanced Cognitive Systems*, 2016.

[80] Sung Wook Yoon, Alan Fern, and Robert Givan. Ff-replan: A baseline for probabilistic planning. In *International Conference on Automated Planning and Scheduling (ICAPS)*, volume 7, pages 352–359, 2007.

Appendix A: The blocksworld domain

The blocksworld domain description used by the MIDCA simulator (we used Pyhop planner for this domain):

```
type(BLOCK)
type(ARSONIST)
type(FIRE-EXTINGUISHER)



predicate(clear, [blk], [BLOCK])
predicate(holding, [blk], [BLOCK])
predicate(arm-empty, [], [])
predicate(on, [blk1, blk2], [BLOCK, BLOCK])
predicate(on-table, [blk], [BLOCK])
predicate(block, [blk], [BLOCK])
predicate(triangle, [blk], [BLOCK])
predicate(table, [blk], [BLOCK])
predicate(onfire, [blk], [BLOCK])
predicate(arsonist, [ars], [ARSONIST])
predicate(free, [ars], [ARSONIST])
predicate(fire-extinguisher, [fireExt], [FIRE-EXTINGUISHER])
predicate(holdingextinguisher, [fireExt],
[FIRE-EXTINGUISHER])
predicate(ext_in_box, [fireExt, blk],
```

```
[FIRE-EXTINGUISHER, BLOCK])


operator(stack,
args = [(topblk, BLOCK), (btmblk, BLOCK)],
preconditions = [
condition(clear, [btmblk]),
condition(holding, [topblk])],
results = [
condition(clear, [btmblk], negate = True),
condition(holding, [topblk], negate = True),
condition(clear, [topblk]),
condition(on, [topblk, btmblk]),
condition(arm-empty, [])])

operator(unstack,
args = [(topblk, BLOCK), (btmblk, BLOCK)],
preconditions = [
condition(clear, [topblk]),
condition(arm-empty),
condition(on, [topblk, btmblk])],
results = [
condition(clear, [topblk], negate = True),
condition(holding, [topblk]),
condition(arm-empty, [], negate = True),
condition(on, [topblk, btmblk], negate = True),
condition(clear, [btmblk], negate = False)])

operator(putdown,
args = [(blk, BLOCK)],
```

```
preconditions = [
condition(holding, [blk])],
results = [
condition(holding, [blk], negate = True),
condition(clear, [blk]),
condition(on-table, [blk]),
condition(arm-empty, [])])

operator(pickup,
args = [(blk, BLOCK)],
preconditions = [
condition(on-table, [blk]),
condition(clear, [blk]),
condition(arm-empty, [])],
results = [
condition(holding, [blk]),
condition(clear, [blk], negate = True),
condition(on-table, [blk], negate = True),
condition(arm-empty, [], negate = True)])

operator(pickup_extinguisher,
args = [(fireExt, FIRE-EXTINGUISHER)],
preconditions = [
condition(holdingextinguisher, [fireExt], negate = True)],
results = [
condition(holdingextinguisher, [fireExt])])

operator(putdown_extinguisher,
args = [(fireExt, FIRE-EXTINGUISHER)],
preconditions = [
```

```
condition(holdingextinguisher, [fireExt])],
results = [
condition(holdingextinguisher, [fireExt], negate = True)])

operator(putoutfire,
args = [(blk, BLOCK), (fireExt, FIRE-EXTINGUISHER)],
preconditions = [
condition(onfire, [blk]),
condition(holdingextinguisher, [fireExt])],
results = [
condition(onfire, [blk], negate = True)])

operator(catchfire,
args = [(blk, BLOCK)],
preconditions = [
condition(onfire, [blk], negate = True)],
results = [
condition(onfire, [blk])])

operator(lightonfire,
args = [(arsonist, ARSONIST), (blk, BLOCK)],
preconditions = [
condition(onfire, [blk], negate = True),
condition(free, [arsonist])],
results = [
condition(onfire, [blk])])

operator(apprehend,
args = [(arsonist, ARSONIST)],
preconditions = [
```

```
condition(free, [arsonist])],
results = [
condition(free, [arsonist], negate = True)])

operator(searchfor,
args = [(arsonist, ARSONIST)])
```

Appendix B: The blocksworld domain for the Baxter robot

Operators and methods definition for the Pyhop planner for the blocksworld domain with the Baxter robot:

Operators definition:

```
def grab(state, objectID):
return state

def release(state, objectID):
return state

def putdown(state,b):
    if get_last_position(state, b) == 'in-arm':
        set_position(state, b, 'table')
        set_clear_status(state, b, 'clear')
        return state
    else: return False

def raising(state, objectID):
return state

def raising_arm(state):
```

```
return state

def reach_to_pickup(state,b):
    if get_last_position(state, b) == 'table' and
    get_last_clear_status(state, b) == 'clear':
        #state.position[b] = 'hand'
        set_position(state, b, 'in-arm')
        #state.isclear[b] = False
        set_clear_status(state, b, 'not clear')
        #state.holding = b
        return state
    else: return False

def reach_to_unstack(state,b,c):
    if get_last_position(state, b) == c and
    get_last_clear_status(state, b) == 'clear':
        set_position(state, b, 'in-arm')
        set_clear_status(state, b, 'not clear')
        set_clear_status(state, c, 'clear')
        return state
    else: return False

def stack(state,b,c):
    if get_last_clear_status(state, c) == 'clear':
        set_position(state, b, c)
        set_clear_status(state, b, 'clear')
        set_clear_status(state, c, 'not clear')
        return state
    else:
     print("return false")
```

```
        return False


def get_last_position(state, objectOrID):
    positions = state.all_pos(objectOrID)
    if not positions:
        return None
    else:
        for state_pos in reversed(positions):
            if state_pos.position:
                return (state_pos.position)
        return None

def set_position(state, objectOrID, newPos):
    positions = state.all_pos(objectOrID)
    if not positions:
        return None
    else:
        for state_pos in reversed(positions):
            if state_pos.position:
                state_pos.position = newPos
                return state
    return None

def get_last_clear_status(state, objectOrID):
    positions = state.all_pos(objectOrID)
    if not positions:
        return None
    else:
        for state_pos in reversed(positions):
```

```
            if state_pos.isclear:
                return (state_pos.isclear)
    return None

def set_clear_status(state, objectOrID, newClearStatus):
    positions = state.all_pos(objectOrID)
    if not positions:
        return None
else:
    for state_pos in reversed(positions):
        if state_pos.isclear:
            state_pos.isclear = newClearStatus
            return state
    return None
```

Methods definition:

```
def point_at_m(state, objectID):
    return [("block_until_seen", objectID),
    ("point_to", objectID)]

def pickup_m(state, objectID):
    return [("reach_to_pickup", objectID), ("grab", objectID),
    ("raising", objectID)]

def unstack_m(state, b1):
    return [("reach_to_unstack", b1,
```

```
    get_last_position(state, b1)),
    ("grab", b1), ("raising", b1)]

def get_last_position(state, objectOrID):
    positions = state.all_pos(objectOrID)
    if not positions:
        return None
    else:
        for state_pos in reversed(positions):
            if state_pos.position:
                return (state_pos.position)
    return None


def get_last_clear_status(state, objectOrID):
    positions = state.all_pos(objectOrID)
    if not positions:
        return None
    else:
        for state_pos in reversed(positions):
            if state_pos.isclear:
                return (state_pos.isclear)
    return None


def all_blocks(state):
    # return state.all_objects()
    return ['green block', 'red block']
```

```
def achieve_goals_m(state, goals):
    print("achieve_goals_m")

    if goals:
        goal = goals[0]
        get_goal_pos(goal)
        object = goal["directObject"]
        if goal["objective"] == "show-loc":
            return [("point_at", goal["directObject"]),
            ("achieve_goals", goals[1:])]

        if goal["objective"] == "stacking":
            print("holding")
            return [("move_blocks", goal)]

        if goal["objective"] == "holding":
            if get_last_clear_status(state, object) == 'clear':
                return [("pickup_task", goal["directObject"]),
                ("achieve_goals", goals[1:])]

        if goal["objective"] == "moving":
            if get_last_clear_status(state, object) == 'clear':
                return [('move_one',
                goal["directObject"], 'table'),
                ("achieve_goals", goals[1:])]

        else:
            print("null")
            return False  # fail if goal is not of known type
    return []  # return empty plan if no goals.
```

```
"""
Here are some helper functions that are used in the methods'
preconditions.
"""
goal_pos_dic = {}

def get_goal_pos(goal):
    poses = goal["pos"]
    goal_pos_dic.update({poses.split(":")[0]:
    poses.split(":")[1]})



def is_done(b1, state, goal):
    if b1 == 'table': return True
    if b1 in goal_pos_dic:
        print((goal_pos_dic[b1]))
    if b1 in goal_pos_dic and
    str(goal_pos_dic[b1]) !=
    str(get_last_position(state, b1)):
        print("return false")
        return False
    if get_last_position(state, b1) == 'table':
    return True
    if get_last_position(state, b1) in
    list(goal_pos_dic.values())
    and (b1 not in goal_pos_dic or
    goal_pos_dic[b1] !=
    get_last_position(state, b1)):
```

```
        return False

    return is_done(get_last_position(state, b1), state, goal)


def status(b1, state, goal):
    if b1 in goal_pos_dic:
        print((goal_pos_dic[b1]))
    if is_done(b1, state, goal):
        return 'done'
    elif not (get_last_clear_status(state, b1) or
    get_last_position(state, b1) == "hand"):
        return 'inaccessible'
    elif not (b1 in goal_pos_dic) or
    str(goal_pos_dic[b1]).strip() == 'table':
        return 'move-to-table'
    elif is_done(goal_pos_dic[b1], state, goal) and
    get_last_clear_status(state, goal_pos_dic[b1]):
        return 'move-to-block'
    else:
        return 'waiting'


"""
In each Pyhop planning method, the first argument is the
current state (this is analogous to Python methods, in which the first
argument is the class instance). The rest of the arguments must
match the arguments of the task that the method is for.
For example, ('pickup', b1) has a method get_m(state,b1),
as shown below.
```

```
"""

### methods for "move_blocks"

def moveb_m(state, goal):
    """
    This method implements the following block-stacking
    algorithm: If there's a block that can be moved
    to its final position, then do so and call
    move_blocks recursively. Otherwise, if there's a
    block that needs to be moved and can be moved to
    the table, then do so and call move_blocks
    recursively. Otherwise, no blocks need to be moved.
    """

    for b1 in all_blocks(state):
        print(("___block: " + b1))
        input('Enter ...')
        s = status(b1, state, goal)
        if s == 'move-to-table':
            print("___move one")
            return [('move_one', b1, 'table'),
            ('move_blocks', goal)]
        elif s == 'move-to-block':
            return [('move_one', b1, goal_pos_dic[b1]),
            ('move_blocks', goal)]
        else:
            print("continue")
            continue
    #
```

```
    # if we get here, no blocks can be moved to their final
    locations
    b1 = pyhop.find_if(lambda x: status(x, state, goal) ==
    'waiting', all_blocks(state))
    if b1 != None:
        return [('move_one', b1, 'table'), ('move_blocks', goal)]
    #
    # if we get here, there are no blocks that need moving
    return []


"""
declare_methods must be called once for each taskname. Below,
'declare_methods('get',get_m)' tells Pyhop that 'get'
has one method, get_m. Notice that 'get' is a quoted
string, and get_m is the actual function.
"""


### methods for "move_one"

def move1(state, b1, dest):
    """
    Generate subtasks to get b1 and put it at dest.
    """
    if get_last_position(state, b1) == "in-arm":
        return [('put', b1, dest)]
    else:
        return [('get', b1), ('put', b1, dest)]
```

```
### methods for "get"

def get_by_unstack(state, b1):
    """Generate a pickup subtask."""
    # if get_last_clear_status(state, b1) == 'clear':
    return [('unstack_task', b1)]
    # return False


def get_m(state, b1):
    """Generate a pickup subtask."""
    if get_last_clear_status(state, b1) == 'clear' and
    get_last_position(state, b1) == 'table':
        return [('pickup_task', b1)]
    elif get_last_clear_status(state, b1) == 'clear' and
    get_last_position(state, b1) != 'table':
        return [('unstack_task', b1)]

    # return False


def put_m(state, b1, b2):
    """
    Generate either a putdown or a stack subtask for b1.
    b2 is b1's destination: either the table or another block.
    """
    if get_last_position(state, b1) == 'in-arm':
        if b2 == 'table':
            return [('putdown', b1), ('release', b1),
```

```
            ('raising', b1)]
        else:
            return [('stack', b1, b2), ('release', b1),
            ('raising', b1)]
    else:
        return False


def put_out_m(state, b1):
    if state.fire[b1]:
        return [("putoutfire", b1)]
    else:
        return []


def quick_apprehend_m(state, perp):
    if state.free[perp]:
        return [("apprehend", perp)]
    else:
        return []


def long_apprehend_m(state, perp):
    if state.free[perp]:
        return [("searchfor", perp), ("searchfor", perp),
        ("searchfor", perp), ("searchfor", perp),
        ("apprehend", perp)]
    else:
        return []
```

Appendix C: The logistics Domain

The logistics domain description used by the JSHOP Planner:

```
(defdomain neo-trans
  (
     (:operator (!load-truck ?obj ?truck ?loc)
                ();;; preconditions
                ((obj-at ?obj ?loc));;; delete list
                ((in-truck ?obj ?truck)));;; add list

     (:operator (!unload-truck ?obj ?truck ?loc)
                ()
                ((in-truck ?obj ?truck))
                ((obj-at ?obj ?loc)))



     (:operator (!load-airplane ?obj ?airplane ?loc)
                ()
                ((obj-at ?obj ?loc))
                ((in-airplane ?obj ?airplane)));;

     (:operator (!unload-airplane ?obj ?airplane ?loc)
                ()
```

```
                ((in-airplane ?obj ?airplane))
                ((obj-at ?obj ?loc)))

        (:operator (!drive-truck ?truck ?locfrom ?locto)
                ()
                ((truck-at ?truck ?locfrom))
                ((truck-at ?truck ?locto)));;

        (:operator (!fly-airplane ?airplane ?airportfrom
        ?airportto)
                ()
                ((airplane-at ?airplane ?airportfrom))
                ((airplane-at ?airplane ?airportto)))


        (:method (achieve-goals (list ?goal .
        ?goals))
                ()
                ((achieve-goal ?goal)(achieve-goals list ?goals)))

        (:method (achieve-goals nil ?goal)
                ()
                ((achieve-goal ?goal)))




;;; Assumes that the destination location and
starting location
;;; are in same city assumes there is a truck in
```

```
starting location
      (:method (achieve-goal (obj-at ?p ?l))
         ((obj-at ?p ?l1) ;;;; preconditions
          (truck-at ?t ?l1)
          (sameCity ?l1 ?l)
         )
         ((!load-truck ?p ?t ?l1)  ;;;; subtasks
          (!drive-truck ?t ?l1 ?l)
      (!unload-truck ?p ?t ?l)
         )
       )

   (:method (achieve-goal (obj-at ?p ?l))
         ((obj-at ?p ?l1) ;;;; preconditions
          (same ?l1 ?l)
         )
         (
         )
       )
;;; Assumes that the destination location and starting
location ;;; are in same city
;;; assumes there is no truck in starting location
(:method (achieve-goal (obj-at ?p ?l))
         ((obj-at ?p ?l1)
          (truck-at ?t ?l2)
          (sameCity ?l2 ?l)
          (sameCity ?l1 ?l)
          (different ?l2 ?l1)
         )
         ((!drive-truck ?t ?l2 ?l1)
```

```
      (achieve-goal (obj-at ?p ?l))
         )
       )


;;; Assumes destination and starting location are in
different cities
;;; assumes package IS AT AIRPORT
;;; assumes there is an airplane in the starting location
(:method (achieve-goal (obj-at ?p ?l))
         ((obj-at ?p ?ap1)
          (differentCity ?l2 ?l1)
          (differentCity ?l1 ?l2)
          (airplane-at ?a1 ?ap1)
          (not (sairplane-at ?a2 ?ap1))
          (AIRPLANE ?a1)
          (sameCity ?ap1 ?l1) ;; airport is in this city
          (sameCity ?l1 ?ap1)
          (sameCity ?l2 ?l)
          (sameCity ?l ?l2)
          (LO ?ap2)
          (sameCity ?ap2 ?l)
          (NearByAirport ?ap1 ?ap2)
         )
         (
          (!load-airplane ?p ?a1 ?ap1)
          (!fly-airplane ?a1 ?ap1 ?ap2)
          (!unload-airplane ?p  ?a1 ?ap2)
          (!fly-airplane ?a1 ?ap2 ?ap1)
          (achieve-goal (obj-at ?p ?l))
```

```
          )
         )
;;3
(:method (achieve-goal (obj-at ?p ?l))
         ((obj-at ?p ?ap1)
          (differentCity ?l2 ?l1)
          (differentCity ?l1 ?l2)
          (airplane-at ?a1 ?ap1)
          (not (sairplane-at ?a2 ?ap1))
          (AIRPLANE ?a1)
          (sameCity ?ap1 ?l1) ;; airport is in this city
          (sameCity ?l1 ?ap1)
          (sameCity ?l2 ?l)
          (sameCity ?l ?l2)
          (LOCATION  ?ap2)
          (sameCity ?ap2 ?l)
          (NearByAirport ?ap1 ?ap3)
          (different ?ap3 ?ap2)
         )
         (
          (!load-airplane ?p ?a1 ?ap1)
          (!fly-airplane ?a1 ?ap1 ?ap3)
          (!unload-airplane ?p  ?a1 ?ap3)
          (!fly-airplane ?a1 ?ap3 ?ap1)
          (achieve-goal (obj-at ?p ?l))
          )
         )
;;4
 ;;;Super plane
  (:method (achieve-goal (obj-at ?p ?l))
```

```
        ((obj-at ?p ?ap1)
         (differentCity ?l2 ?l1)
         (differentCity ?l1 ?l2)
         (sairplane-at ?a1 ?ap1)
         (SAIRPLANE ?a1)
         (sameCity ?ap1 ?l1) ;; airport is in this city
         (sameCity ?l1 ?ap1)
         (sameCity ?l2 ?l)
         (sameCity ?l ?l2)
         (LOCATION  ?ap2)
         (sameCity ?ap2 ?l)

        )
        (
         (!load-airplane ?p ?a1 ?ap1)
         (!fly-airplane ?a1 ?ap1 ?ap2)
         (!unload-airplane ?p  ?a1 ?ap2)
         (achieve-goal (obj-at ?p ?l))
         )
        )

;;; Assumes destination and starting location are in
different cities
;;; assumes package IS IN SAME CITY BUT NOT AT AIRPORT
;;; assumes there is an airplane in the starting location
(:method (achieve-goal (obj-at ?p ?l))
        ((obj-at ?p ?l1)
         (differentCity ?l2 ?l1)
         (differentCity ?l1 ?l2)
         (airplane-at ?a1 ?ap1)
```

```
          (AIRPLANE ?a1)
          (not (sairplane-at ?a2 ?ap1))
          (sameCity ?ap1 ?l1) ;; airport is in this city
          (sameCity ?l1 ?ap1)
          (sameCity ?l2 ?l)
          (sameCity ?l ?l2)
          (LOCATION  ?ap2)
          (sameCity ?ap2 ?l)
          (NearByAirport ?ap1 ?ap2)
         )
         ((achieve-goal (obj-at ?p ?ap1))
          (!load-airplane ?p ?a1 ?ap1)
          (!fly-airplane ?a1 ?ap1 ?ap2)
          (!unload-airplane ?p  ?a1 ?ap2)
          (!fly-airplane ?a1 ?ap2 ?ap1)
          (achieve-goal (obj-at ?p ?l))
          )
         )

(:method (achieve-goal (obj-at ?p ?l))
         ((obj-at ?p ?l1)
          (differentCity ?l2 ?l1)
          (differentCity ?l1 ?l2)
          (airplane-at ?a1 ?ap1)
          (not (sairplane-at ?a2 ?ap1))
          (AIRPLANE ?a1)
          (sameCity ?ap1 ?l1) ;; airport is in this city
          (sameCity ?l1 ?ap1)
          (sameCity ?l2 ?l)
          (sameCity ?l ?l2)
```

```
      (LOCATION  ?ap2)
      (sameCity ?ap2 ?l)
      (NearByAirport ?ap1 ?ap3)
      (different ?ap3 ?ap2)
     )
     ((achieve-goal (obj-at ?p ?ap1))
      (!load-airplane ?p ?a1 ?ap1)
      (!fly-airplane ?a1 ?ap1 ?ap3)
      (!unload-airplane ?p  ?a1 ?ap3)
      (!fly-airplane ?a1 ?ap3 ?ap1)
      (achieve-goal (obj-at ?p ?l))
      )
     )

(:method (achieve-goal (obj-at ?p ?l))
     ((obj-at ?p ?l1)
      (differentCity ?l2 ?l1)
      (differentCity ?l1 ?l2)
      (sairplane-at ?a1 ?ap1)
      (SAIRPLANE ?a1)
      (sameCity ?ap1 ?l1) ;; airport is in this city
      (sameCity ?l1 ?ap1)
      (sameCity ?l2 ?l)
      (sameCity ?l ?l2)
      (LOCATION  ?ap2)
      (sameCity ?ap2 ?l)

     )
     ((achieve-goal (obj-at ?p ?ap1))
      (!load-airplane ?p ?a1 ?ap1)
```

```
          (!fly-airplane ?a1 ?ap1 ?ap2)
          (!unload-airplane ?p  ?a1 ?ap2)
          (achieve-goal (obj-at ?p ?l))
          )
         )

;; Assumes destination and starting location are in
different cities
;; assumes there is NOT an airplane in the starting
location,
;; instead there is an airplane in another city
(:method (achieve-goal (obj-at ?p ?l))
         ((obj-at ?p ?l1)
          (differentCity ?l2 ?l1)
          (differentCity ?l1 ?l2)
          (airplane-at ?a1 ?ap3)
          (LOCATION  ?ap2)
          (LOCATION  ?ap3)
          (sameCity ?l2 ?l)
          (sameCity ?l ?l2)
          (sameCity ?ap2 ?l1)
          (sameCity ?l1 ?ap2)
          (different ?ap2 ?ap3)
         )
         ((achieve-goal (obj-at ?p ?ap2))
          (!fly-airplane ?a1 ?ap3 ?ap2)
          (achieve-goal (obj-at ?p ?l))
          )
         )
```

```
;;;-------------------------------------


    (:- (same ?x ?x) ())
    (:- (sameCity ?a ?b)
    ((IN-CITY ?a ?c) (IN-CITY ?b ?c))
    )
(:- (NearByAirport ?a ?b)
    ((LOCATION  ?a)(LOCATION ?b)(IN-CITY ?a ?c) (IN-CITY ?b ?d)
    (NearBy ?c ?d))
    )

    (:- (differentCity ?a ?b)
    ((IN-CITY ?a ?c) (IN-CITY ?b ?d) (different ?c ?d))
    )
    (:- (different ?x ?y) ((not (same ?x ?y))))

    (:- (vehicle ?v) ((smallVehicle ?v)))
    (:- (vehicle ?v) ((bigVehicle ?v)))
    (:- (transport ?x) ((vehicle ?x)))
  ))
```

Appendix D: The logistics domain (the package delivery domain)

```
(defdomain logistics
   (
    (:operator (!load-truck ?obj ?truck ?loc)
               ()
               ((obj-at ?obj ?loc) (:protection (truck-at
               ?truck ?loc)))
               ((in-truck ?obj ?truck)))

    (:operator (!unload-truck ?obj ?truck ?loc)
               ()
               ((in-truck ?obj ?truck) (:protection
               (truck-at ?truck ?loc)))
               ((obj-at ?obj ?loc)))

    (:operator (!load-airplane ?obj ?airplane ?loc)
               ()
               ((obj-at ?obj ?loc) (:protection (airplane-at
               ?airplane ?loc)))
               ((in-airplane ?obj ?airplane)))

    (:operator (!unload-airplane ?obj ?airplane ?loc)
```

```
            ()
            ((in-airplane ?obj ?airplane) (:protection
            (airplane-at ?airplane ?loc)))
            ((obj-at ?obj ?loc)))

(:operator (!drive-truck ?truck ?loc-from ?loc-to)
            ()
            ((truck-at ?truck ?loc-from))
            ((truck-at ?truck ?loc-to) (:protection
            (truck-at ?truck ?loc-to))))

(:operator (!fly-airplane ?airplane ?airport-from
?airport-to)
            ()
            ((airplane-at ?airplane ?airport-from))
            ((airplane-at ?airplane ?airport-to)
            (:protection (airplane-at ?airplane
            ?airport-to))))


(:operator (!add-protection ?g)
            ()
            ()
            ((:protection ?g))
            )

(:operator (!delete-protection ?g)
            ()
            ((:protection ?g))
            ()
```

```
           )

(:method (obj-at ?obj ?loc-goal)
         ((in-city ?loc-goal ?city-goal)
          (obj-at ?obj ?loc-now)
          (in-city ?loc-now ?city-goal)
          (truck ?truck ?city-goal)
          )
  ((in-city-delivery ?truck ?obj ?loc-now ?loc-goal))

  ((in-city ?loc-goal ?city-goal)
   (obj-at ?obj ?loc-now)
   (in-city ?loc-now ?city-now)
   (different ?city-goal ?city-now)
   (truck ?truck-now ?city-now)
   (truck ?truck-goal ?city-goal)
   (airport ?airport-now)
   (in-city ?airport-now
   ?city-now)
   (airport ?airport-goal)
   (in-city ?airport-goal ?city-goal))
   ((in-city-delivery ?truck-now ?obj ?loc-now
   ?airport-now)
   (air-deliver-obj ?obj
   ?airport-now ?airport-goal)
    (in-city-delivery ?truck-goal ?obj ?airport-goal
    ?loc-goal)))

(:method (in-city-delivery ?truck ?obj ?loc-from
?loc-to)
```

```
          ((same ?loc-from ?loc-to))
          ()

          ((in-city ?loc-from ?city)
           (truck ?truck ?city))
          ((truck-at ?truck ?loc-from)
          (:immediate !load-truck ?obj ?truck ?loc-from)
          (truck-at ?truck ?loc-to)
          (:immediate !unload-truck ?obj ?truck
          ?loc-to)))


(:method (truck-at ?truck ?loc-to)

          ((truck-at ?truck ?loc-from)
           (different ?loc-from ?loc-to))
          ((:immediate !drive-truck ?truck
          ?loc-from ?loc-to))

          ((truck-at ?truck ?loc-from)
           (same ?loc-from ?loc-to))
          ((:immediate !add-protection
          (truck-at ?truck ?loc-to))))

(:method (air-deliver-obj ?obj ?airport-from ?airport-to)
         airplane-at-the-current-airport
          ((airplane-at ?airplane ?airport-from))
          ((:immediate !add-protection (airplane-at
          ?airplane ?airport-from))
         (!load-airplane ?obj ?airplane ?airport-from)
```

```
        (fly-airplane ?airplane ?airport-to)
        (!unload-airplane ?obj ?airplane ?airport-to))

         ((airplane-at ?airplane ?any-airport))
         ((:immediate !fly-airplane ?airplane
         ?any-airport ?airport-from)
        (!load-airplane ?obj ?airplane ?airport-from)
        (fly-airplane ?airplane ?airport-to)
        (!unload-airplane ?obj ?airplane ?airport-to)))

(:method (fly-airplane ?airplane ?airport-to)
         airplane-already-there
         ((airplane-at ?airplane ?airport-to))
         ((:immediate !add-protection (airplane-at
         ?airplane ?airport-to)))

         fly-airplane-in
         ((airplane-at ?airplane ?airport-from))
         ((:immediate !fly-airplane ?airplane
         ?airport-from ?airport-to)))


(:- (same ?x ?x) nil)
(:- (different ?x ?y) ((not (same ?x ?y))))
))
```

Appendix E: The minecraft domain

The PDDL planning file for the minecraft domain used by the Metric-FF planner:

```
(define (domain minecraft-beta)
    (:requirements :typing :fluents
     :existential-preconditions)
    (:types
        resource - thing
        material - thing
        tool - thing
        craftgrid
        mapgrid
        direction
        player
        monster - thing
        weapon - thing
        potion - thing
        helmet - thing
        chestplate - thing
    )

    (:predicates
        (player-at  ?loc - mapgrid)
        (in-shelter)
```

```
(trap-destroyed ?loc - mapgrid)
(searched-left ?obj - resource ?loc - mapgrid)
(searched-right ?obj - resource ?loc - mapgrid)
(searched-behind ?obj - resource ?loc - mapgrid)
(searched-forward ?obj - resource ?loc - mapgrid)
(looking-right  ?loc - mapgrid)
(looking-left  ?loc - mapgrid)
(looking-forward  ?loc - mapgrid)
(looking-behind  ?loc - mapgrid)
(thing-at-map  ?obj - resource  ?loc - mapgrid)
(thing-at ?obj - resource ?loc - mapgrid)
(known-loc ?obj - resource ?playerloc - mapgrid )
(thing-at-loc ?obj - resource ?loc - mapgrid)
(placed-thing-at-map  ?obj - material  ?loc - mapgrid)
(resource-at-craft  ?res - thing  ?loc - craftgrid)
(craft-empty  ?loc - craftgrid)
(connect  ?from - mapgrid  ?to - mapgrid)
(connect-left  ?from - mapgrid  ?to - mapgrid)
(connect-right  ?from - mapgrid  ?to - mapgrid)
(connect-behind  ?from - mapgrid  ?to - mapgrid)
(connect-forward  ?from - mapgrid  ?to - mapgrid)
(know-where ?res - resource ?loc - mapgrid)
(crafting)
(survive)
(attacking ?loc - mapgrid)
(looking-for ?res - resource ?loc - mapgrid)
(head-armed)
(chest-armed)
(is-attacked ?loc - mapgrid)
(is-trapped ?loc - mapgrid)
```

```
)

(:functions
    (thing-available  ?obj - thing)
    (current-harvest-duration)
    (current-harvest-location)
    (duration-need  ?tool - tool  ?res - resource)
    (location-id  ?loc - mapgrid)
    (tool-id  ?tool - tool)
    (tool-in-hand)
    (tool-max-health ?tool - tool)
    (tool-current-health ?tool - tool)
    (player-current-health)
    (current-hunger-value)

)

;; ---------------------------------------------------

(:action place-on-map
    :parameters (?res - material  ?target - mapgrid)
    :precondition
        (and

            (player-at ?target)
            (not (placed-thing-at-map ?res ?target))
            (> (thing-available ?res) 0)
        )
    :effect
        (and
```

```
            (placed-thing-at-map ?res ?target)
            (decrease (thing-available ?res) 1)
            (assign (current-harvest-duration) 0)
        )
)

;; ------------------------------------------------
(:action move
    :parameters (?from - mapgrid ?to - mapgrid)
    :precondition
        (and
         (> (player-current-health) 0)
            (player-at ?from)
            (connect ?from ?to)
        )
    :effect
        (and
            (player-at ?to)
            (not (player-at ?from))
            (assign (current-harvest-duration) 0)
            (assign (current-harvest-location) 0)
        )
)
;;

;;---------------------------------------
(:action find-forward
    :parameters (?res -resource ?playerloc - mapgrid )
    :precondition
        (and
```

```
            (> (player-current-health) 0)
            (player-at ?playerloc)
            (not (known-loc ?res ?playerloc))
        )
    :effect
        (and
            (searched-forward ?res ?playerloc)
            (looking-forward ?playerloc)
        )
)
;-------------------------------------------------------
(:action find-left
    :parameters (?res -resource ?playerloc - mapgrid )
    :precondition
        (and
            (> (player-current-health) 0)
            (searched-forward ?res ?playerloc)
            (player-at ?playerloc)
            (not (known-loc ?res ?playerloc))
        )
    :effect
        (and
            (searched-left ?res ?playerloc)
            (not (looking-forward ?playerloc))
            (looking-left ?playerloc)
        )
)

;;---------------------------------------
```

```
(:action find-right
    :parameters (?res -resource ?playerloc - mapgrid )
    :precondition
        (and
            (> (player-current-health) 0)
            (searched-left ?res ?playerloc)
            (player-at ?playerloc)
            (not (known-loc ?res ?playerloc))
        )
    :effect
        (and
            (searched-right ?res ?playerloc)
            (not (looking-left ?playerloc))
            (looking-right ?playerloc)
        )
)
;;---------------------------------------

(:action find-behind
    :parameters (?res -resource ?playerloc - mapgrid )
    :precondition
        (and
            (> (player-current-health) 0)
           (searched-right ?res ?playerloc)
           (player-at ?playerloc)
            (not (known-loc ?res ?playerloc))
        )
    :effect
        (and
            (known-loc ?res ?playerloc)
```

```
            (not (looking-right ?playerloc))
            (looking-behind ?playerloc)
            (looking-for ?res ?playerloc)
        )
)

;;-------------------------------------------------------
(:action attack-skeleton
    :parameters (?tool - tool ?loc - mapgrid)
    :precondition
        (and
            (head-armed)
            (> (player-current-health) 0)
            (player-at ?loc)
            (known-loc skeleton ?loc)
            (thing-at skeleton ?loc)
            (= (tool-id ?tool) 10)
            (= (tool-in-hand) 10)
        )
    :effect
        (and
            (not (thing-at skeleton ?loc))
            (attacking ?loc)
        )
)

;;---------------------------------
(:action destroy-trap-with-loc
    :parameters (?tool - tool ?loc - mapgrid
    ?player_loc - mapgrid)
```

```
        :precondition
            (and
                (chest-armed)
                (> (player-current-health) 0)
                (thing-at-map arrowtrap ?loc)
                (= (tool-id ?tool) 11)
                (= (tool-in-hand) 11)
                (player-at ?player_loc)
            )
        :effect
            (and

                (not (thing-at-map arrowtrap ?loc))
                (trap-destroyed ?player_loc)
            )
    )
;;

(:action attack-skeleton-with-loc
        :parameters (?tool - tool ?loc - mapgrid
        ?player_loc - mapgrid)
        :precondition
            (and
                (> (player-current-health) 0)
                (thing-at-map skeleton ?loc)
                (head-armed)
                (= (tool-id ?tool) 10)
                (= (tool-in-hand) 10)
                (player-at ?player_loc)
            )
```

```
    :effect
        (and

            (not (thing-at-map skeleton ?loc))
            (attacking ?player_loc)
        )
)

;;-------------------------------------------------------
(:action destroy-trap
    :parameters (?tool - tool ?loc - mapgrid)
    :precondition
        (and
            (chest-armed)
            (> (player-current-health) 0)
            (player-at ?loc)
            (known-loc arrowtrap ?loc)
            (thing-at arrowtrap ?loc)
            (= (tool-id ?tool) 11)
            (= (tool-in-hand) 11)
        )
    :effect
        (and
            (not (thing-at arrowtrap ?loc))
            (trap-destroyed ?loc)
        )
)
;-------------------------------------------------------
```

```
(:action change-harvest-loc
    :parameters (?target - mapgrid)
    :precondition
        (and
            (> (player-current-health) 0)
            (not (= (current-harvest-location)
            (location-id ?target)))
            (player-at ?target)
        )
    :effect
        (and
            (assign (current-harvest-location)
            (location-id ?target))
        )
)

;; ----------------------------------------------------
(:action change-harvest-tool
    :parameters (?tool - tool)
    :precondition
        (and
         (> (player-current-health) 0)
            (not (= (tool-in-hand) (tool-id ?tool)))
            (> (thing-available ?tool) 0)
        )
    :effect
        (and
            (assign (current-harvest-duration) 0)
            (assign (tool-in-hand) (tool-id ?tool))
        )
```

```
)
;;-----------------------------------------------

(:action move-to-shelter
    :parameters (?target - mapgrid )
    :precondition
        (and
            (player-at ?target)
            (thing-at-map shelter ?target)
            (not (in-shelter))
        )
    :effect
        (and
            (in-shelter)
        )
)
;;-----------------------------------------------------
(:action wear-chestplate
    :parameters (?chestplate - chestplate )
    :precondition
        (and
         (> (player-current-health) 0)
            (> (thing-available ?chestplate) 0)
        )
    :effect
        (and
            (chest-armed)
        )
)
;;-----------------------------------------------
```

```
(:action wear-helmet
    :parameters (?helmet - helmet )
    :precondition
        (and
         (> (player-current-health) 0)
            (> (thing-available ?helmet) 0)
        )
    :effect
        (and
            (head-armed)
        )
)

;; ------------------------------------------------------
(:action harvest
    :parameters (?target - mapgrid ?tool - tool
    ?obj - resource)
    :precondition
        (and
         (> (player-current-health) 0)
            (player-at ?target)
            (> (thing-available ?tool) 0)
            (thing-at-map ?obj ?target)
            (= (current-harvest-location)
            (location-id ?target))
            (= (tool-id ?tool) 11)
            (= (tool-in-hand) 11)
        )
    :effect
        (and
```

```
            (increase (current-harvest-duration) 1)
            (decrease (tool-current-health ?tool) 1)
        )
)

;; -----------------------------------------------------
(:action harvest-loose-tool
    :parameters (?tool - tool)
    :precondition
        (and
            (> (thing-available ?tool) 0)
            (= (tool-in-hand) (tool-id ?tool))
            (= (tool-current-health ?tool) 0)

        )
    :effect
        (and
            (decrease (thing-available ?tool) 1)
            (assign (tool-current-health ?tool)
            (tool-max-health ?tool))
        )
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Harvesting
;;  tree -> wood
;; rock -> stone
;; coalore -> coal
;; ironore -> ironore
;; tallgrass -> seeds
```

```
;; wheatgrass -> wheat
;; sandrock -> sandstone
;; soil -> sand
;; claysoil -> clay
;; brown-mushroom -> brown-mushroom
;; red-mushroom -> red-mushroom
;; skeleton -> bone
;; sugarcane -> sugar
;; cobweb : shears -> 1 string
;; chicken : hand -> egg
;; water : fishingrod -> fish
;; sheep : shears -> 4 wool
;; cow : bucket -> milk
;; ------------------------------------------------------
(:action get-harvest-wood
    :parameters (?target - mapgrid ?tool - tool)
    :precondition
        (and
        (> (player-current-health) 0)
            (player-at ?target)
            (thing-at-map tree ?target)
            (= (tool-id ?tool) 11)
            (= (tool-in-hand) 11)
            (= (current-harvest-location)
            (location-id ?target))
        )
    :effect
        (and
            (increase (thing-available wood) 1)
            (not (thing-at-map tree ?target))
```

)

)

)